Formal Foundations for Translational Separation Logic Verifiers

THIBAULT DARDINIER, ETH Zurich, Switzerland
MICHAEL SAMMLER, ETH Zurich, Switzerland
GAURAV PARTHASARATHY, ETH Zurich, Switzerland
ALEXANDER J. SUMMERS, University of British Columbia, Canada
PETER MÜLLER, ETH Zurich, Switzerland

Program verification tools are often implemented as front-end translations of an input program into an intermediate verification language (IVL) such as Boogie, GIL, Viper, or Why3. The resulting IVL program is then verified using an existing back-end verifier. A soundness proof for such a *translational verifier* needs to relate the input program and verification logic to the semantics of the IVL, which in turn needs to be connected with the verification logic implemented in the back-end verifiers. Performing such proofs is challenging due to the large semantic gap between the input and output programs and logics, especially for complex verification logics such as separation logic.

This paper presents a formal framework for reasoning about translational separation logic verifiers. At its center is a generic core IVL that captures the essence of different separation logics. We define its operational semantics and formally connect it to two different back-end verifiers, which use symbolic execution and verification condition generation, resp. Crucially, this semantics uses angelic non-determinism to enable the application of different proof search algorithms and heuristics in the back-end verifiers. An axiomatic semantics for the core IVL simplifies reasoning about the front-end translation by performing essential proof steps once and for all in the equivalence proof with the operational semantics rather than for each concrete front-end translation.

We illustrate the usefulness of our formal framework by instantiating our core IVL with elements of Viper and connecting it to two Viper back-ends as well as a front-end for concurrent separation logic. All our technical results have been formalized in Isabelle/HOL, including the core IVL and its semantics, the semantics of two back-ends for a subset of Viper, and all proofs.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Automated reasoning**; **Separation logic**; **Operational semantics**; **Axiomatic semantics**; **Program verification**.

Additional Key Words and Phrases: deductive verifier, translational verifier, intermediate verification language, angelic non-determinism, implicit dynamic frames, Viper

---

Authors' Contact Information: Thibault Dardinier, ETH Zurich, Department of Computer Science, Zurich, Switzerland, thibault.dardinier@inf.ethz.ch; Michael Sammler, ETH Zurich, Department of Computer Science, Zurich, Switzerland, michael.sammler@inf.ethz.ch; Gaurav Parthasarathy, ETH Zurich, Department of Computer Science, Zurich, Switzerland, gaurav.parthasarathy@inf.ethz.ch; Alexander J. Summers, University of British Columbia, Vancouver, Canada, alex.summers@ubc.ca; Peter Müller, ETH Zurich, Department of Computer Science, Zurich, Switzerland, peter.mueller@inf.ethz.ch.

---

1 Introduction

Many program verification tools are organized into a *front-end*, which encodes an input program along with its specification and verification logic into an intermediate verification language (IVL), and a *back-end*, which computes proof obligations from the IVL program and discharges them, for instance, using an SMT solver. Examples of such *translational verifiers* include Civl [Kragl and Qadeer 2021] and Dafny [Leino 2010] based on the Boogie IVL [Leino 2008], Creusot [Denis et al. 2022] and Frama-C [Kirchner et al. 2015] based on Why3 [Filliâtre and Paskevich 2013], Gillian for C and JavaScript [Maksimovic et al. 2021a] and Rust [Ayoun et al. 2024] based on GIL [Santos et al. 2020], as well as Prusti [Astrauskas et al. 2019] and VerCors [Blom et al. 2017] based on Viper [Müller et al. 2016b].

Developing a program verifier on top of an IVL has major engineering benefits. Most importantly, back-end verifiers, which often contain complex proof search algorithms, sophisticated optimizations, and functionality to communicate with solvers and to report errors, can be re-used across different verifiers, which reduces the effort of developing a program verifier dramatically.

On the other hand, formal reasoning about translational verifiers, in particular, proving their soundness, is more difficult than for verifiers developed by embedding a program logic in an interactive theorem prover (such as Bedrock [Chlipala 2011], VST [Cao et al. 2018], and RefinedC [Sammler et al. 2021]). Proving that a translational verifier is sound requires (1) a formal semantics of the IVL as well as proofs that connect the IVL program (2) to the verification back-end and (3) to the input program. While these steps have been studied for IVLs based on standard first-order logic [Parthasarathy et al. 2021; Cohen and Johnson-Freyd 2024; Herms 2013], they pose additional challenges for IVLs that natively support more-complex widely-used reasoning principles such as those of separation logic (SL) [Reynolds 2002] (and variations such as implicit dynamic frames (IDF) [Smans et al. 2012]). We focus on these IVLs, which are commonly-used and especially useful for building verifiers for heap-manipulating and concurrent programs.

Challenge 1: Defining the semantics of the IVL. Standard programming languages and the intermediate languages used in compilers come with a notion of execution that can naturally be captured by an operational semantics. In contrast, IVLs are typically not designed to be executable, but instead to capture a wide range of verification problems and algorithms for solving them.

To capture different verification *problems*, IVLs contain features that enable the encoding of a diverse set of input programs (*e.g.*, by offering generic operations suitable for encoding different concurrency primitives), specifications (*e.g.*, by offering rich assertion languages), and verification logics (*e.g.*, by supporting concepts such as framing). An IVL semantics must reflect this generality. For instance, separation logic-based IVLs provide complex primitives for manipulating separation logic resources, which can be used to encode separation logic rules into the IVL. As a result, these primitives can be used to encode a large variety of input program features including procedure calls, loops, and concurrency.

To capture different verification *algorithms*, an IVL semantics must not prescribe *how* to construct a proof and should instead abstract over different algorithms. Back-ends should have the freedom to apply various techniques to compute proof obligations (*e.g.*, symbolic execution or verification condition generation), to resolve trade-offs between completeness and automation (*e.g.*, by over-approximating proof obligations), and to discharge proof obligations (*e.g.*, instantiating existentially quantified variables in different ways). For instance, existing algorithms have different performance characteristics for different classes of verification problems [Eilers et al. 2024]; an IVL semantics should provide the freedom to choose the best one for the problem at hand. In practice, capturing different verification algorithms is important for verifiers with multiple back-ends for the same language (*e.g.*, based on either symbolic execution or verification condition generation). However,

even a single back-end may offer a variety of different algorithms, which are chosen based on heuristics or configured by the user (*e.g.*, via command-line options). Moreover, back-ends along with their verification algorithms apply different algorithms over time, as their developers optimize existing verification algorithms or add support for new verification algorithms.

Challenge 2: Connecting the IVL to back-ends. Soundness requires that successful verification of an IVL program by a back-end verifier implies the correctness of the IVL program. Since a back-end verifier's algorithm ultimately decides the outcome of a verification run, a soundness proof needs to formally connect the concrete verification algorithm to the IVL's semantics. In particular, this soundness proof needs to consider the proof search algorithms and optimizations performed by a concrete verification back-end and show that they produce correct results according to the IVL semantics. However, different back-ends typically use a diverse range of strategies to (for example) represent the program state, unroll recursive definitions, choose existentially-quantified permission amounts, and select the footprints of magic wands [Dardinier et al. 2022b].

Challenge 3: Connecting the IVL to front-ends. Soundness also requires that the correctness of the IVL program implies the correctness of the *input* program with respect to its intended verification logic. Such soundness proofs are difficult due to the large semantic gap between input and IVL programs. The two programs may use different reasoning concepts and proof rules, which need to be connected by a soundness proof. This gap is particularly large for typical encodings into IVLs based on separation logic, because the verification logic for the source of this translation is typically different from the one for the IVL program, e.g., one of the vast wealth of concurrent separation logics. For instance, a parallel composition of two threads in the input program is typically encoded as *three sequential* IVL programs: two for the parallel branches, each of which is verified using a separate specification provided by the user, and one for the enclosing code, which composes the two specifications to encode the behavior of the parallel composition overall. Such a translation of front-end proof rules into multiple sequential verification problems is not obvious; a soundness proof must bridge this gap.

Prior work. Several works formalize aspects of translational verifiers with IVLs based on separation logic, but none of them addresses all three challenges outlined above. For Viper, Parthasarathy et al. [2024] build a proof-producing version of Viper's verification condition generation back-end, but do not attempt to connect it to front-end languages nor give a general semantics for Viper that would also capture Viper's symbolic execution back-end. Similarly, Zimmerman et al. [2024] formalize a version (only) of Viper's *symbolic execution* back-end; their focus is on adapting it to gradual verification. Vogels et al. [2015] show the soundness of the symbolic execution of Veri-Fast [Jacobs et al. 2011] w.r.t. an input C program.[1] However, VeriFast has only a single (symbolic execution) back-end that is used as the basis for multiple front-end languages (C, Java, Rust) and thus the formalization does not abstract over different verification algorithms.

Maksimovic et al. [2021b] briefly describe a soundness framework for GIL [Maksimovic et al. 2021a], a parametric program representation used by the Gillian project. GIL needs to be instantiated with a state model, primitive assertions, and memory actions to obtain specific intermediate representations (essentially, multiple IVLs) useful for different verification projects (e.g., for JavaScript [Maksimovic et al. 2021a] and Rust [Ayoun et al. 2024]). However, each GIL instantiation also determines the back-end verification algorithm. As such, there is no common semantics that abstracts over different verification algorithms.

---

[1]VeriFast itself is not an IVL, but must address similar challenges to IVLs based on separation logic since VeriFast's symbolic execution is used to justify multiple front-end languages (C, Java, Rust) using separation logic reasoning; its symbolic execution also has strong similarities with IVL back-ends.

This work. In this paper, we present a framework for formally justifying translational separation logic verifiers. At its center is a generic IVL, called CoreIVL, that captures the essence of different IVLs based on separation logics. In particular, CoreIVL can be instantiated with different statements, assertion languages, and separation algebras; we use a generalized notion of separation algebra that allows us to also model the implicit dynamic frames logic used in Viper.

To address Challenge 1 above, we define the semantics of CoreIVL (and correspondingly, each of its instantiations) using *dual (i.e., demonic and angelic) non-determinism*. Demonic non-determinism is a standard technique to verify properties for all inputs, thread schedules, etc. Our novel insight is to complement it with angelic non-determinism to abstract over the different proof search algorithms employed by back-ends. Intuitively, the IVL program is correct if *any* of these algorithms succeeds, which is an angelic behavior.

To address Challenge 2, we define an operational semantics for CoreIVL, which incorporates these notions of dual non-determinism and, like CoreIVL itself, is parametric in the separation algebra to support both separation logic and IDF. An *operational* semantics facilitates proving a formal connection to the concrete verification algorithms used in back-ends. Separation logic verifiers typically perform symbolic execution, which is typically described operationally [de Boer and Bonsangue 2021] and (as we show) can be connected to our operational semantics via a standard simulation proof. Similarly, an operational IVL semantics is well-suited for formalizing the connections to back-ends that encode IVL programs into a further, more basic IVL, such as Viper's verification condition generator, which encodes Viper programs into Boogie.

To address Challenge 3, we define an axiomatic semantics for CoreIVL and prove its equivalence to our operational semantics. An *axiomatic* semantics facilitates proving a formal connection to the program logic used on the front-end level because both deal with derivations, which are often structurally related due to the compositional nature of most IVL translations. In addition, we are able to prove some powerful generic results about idiomatic encoding patterns once-and-for-all, further minimizing the instantiation-specific gap that a formal soundness proof needs to bridge.

We illustrate the practical applicability of our formal framework by instantiating CoreIVL with elements of Viper. We use the resulting operational semantics to prove the soundness of two verification back-ends: a formalization of the central features of Viper's symbolic execution back-end, and a pre-existing formalization of Viper's verification condition generator [Parthasarathy et al. 2024]. These proofs demonstrate, in particular, that our use of angelic non-determinism allows us to capture these two rather disparate (and representative) back-ends. At the other end, we prove soundness of a front-end based on concurrent separation logic using our axiomatic semantics. These proofs demonstrate that our framework effectively closes the large semantic gap between front-ends and back-ends and enables formal reasoning about the entire chain.

Contributions and outline. We make the following technical contributions:

- We present a formal framework for reasoning about translational separation logic verifiers, via a parametric language CoreIVL, for which we define a novel operational semantics combining core separation-logic reasoning principles and dual non-determinism. We define an alternative axiomatic semantics, and show its equivalence with our operational semantics.
- We define a Viper instantiation of CoreIVL. We formalize and prove the soundness of the core of Viper's symbolic execution back-end. Similarly, we show soundness of an existing formalization of Viper's back-end based on verification condition generation. These proofs illustrate how angelic non-determinism can abstract over these different algorithmic choices.
- We formalize a front-end for a simple concurrent language to be verified with concurrent separation logic, as well as its standard encoding as employed in translational verifiers, and prove this encoding sound with respect to our axiomatic semantics for CoreIVL.
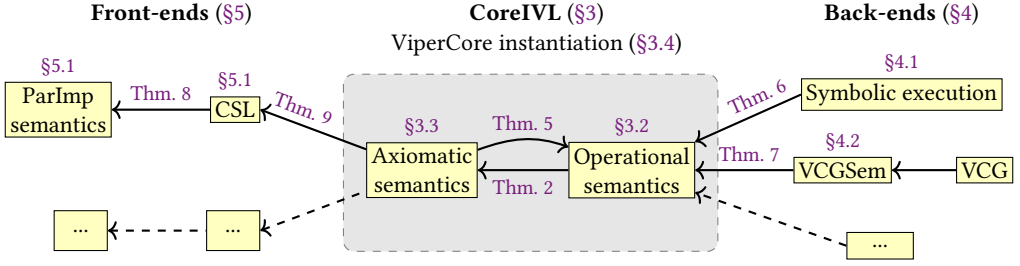
Fig. 1. Overview of our framework and its application to Viper. The yellow boxes represent components of our framework (such as semantics and logics), while the arrows show the theorems that connect them. The dashed arrows and the unlabeled yellow boxes represent potential additional front-ends and back-ends that could be connected to ViperCore. CSL stands for concurrent separation logic, and VCG for verification condition generation. VCG has been formally connected to VCGSem by Parthasarathy et al. [2024].

We give an overview of our key ideas in §2. We define the operational and axiomatic IVL semantics in §3. We discuss how to prove back-end soundness in §4 and front-end soundness in §5. We discuss related work in §6 and conclude in §7.

All formalizations and proofs in this paper are mechanized in the Isabelle proof assistant [Nipkow et al. 2002] and our mechanization is publicly available [Dardinier et al. 2024b].

2 Key Ideas

In this section, we present the key ideas behind our work. Our framework and its instantiation to Viper is presented in Fig. 1. At its center is *CoreIVL* (depicted by the grey area), a general core language for representing SL-based IVLs. This core language bridges the substantial gap between proofs of high-level programs using custom verification logics (*e.g.*, concurrent separation logic [O'Hearn 2004] (CSL) in the figure) at the front-end level and verification algorithms for SL-based IVLs at the back-end level (*e.g.*, symbolic execution and verification condition generation). CoreIVL is *parametric* in its state model and assertions, so that it can represent multiple variants of separation logic (*e.g.*, those on which VeriFast and GIL are based), including implicit dynamic frames (on which Viper is based). In Fig. 1, ViperCore represents the instantiation of these parameters for Viper. We give two equivalent semantics to CoreIVL: An *operational* semantics, which is designed to enable soundness proofs for diverse back-end verification algorithms (shown on the right of Fig. 1), and an *axiomatic* semantics, which can be used to prove front-end translations into CoreIVL sound, by connecting this axiomatic semantics to the front-end separation logic (shown on the left).

The rest of this section is organized as follows. §2.1 introduces the general core language *CoreIVL* for representing SL-based IVLs. §2.2 illustrates how to check for the existence of a Concurrent Separation Logic front-end proof for a parallel program by encoding the verification problem into our sequential CoreIVL, mimicking the approach of modern translational verifiers. §2.3 presents the formal operational semantics of CoreIVL. Finally, §2.4 presents an alternative equivalent *axiomatic semantics* for CoreIVL, and shows how it can be leveraged to prove a front-end translation sound.

2.1 A Core Language for SL-Based IVLs

In this section, we first motivate and then define a core language for SL-based IVLs, called *CoreIVL*, which captures central aspects of SL-based verifiers, such as Viper [Müller et al. 2016b], Gillian [Santos et al. 2020; Maksimovic et al. 2021a], or VeriFast [Jacobs et al. 2011].

$$C ::= \textbf{inhale } A \mid \textbf{exhale } A \mid \textbf{havoc } x \mid C; C \mid \textbf{if}(b) \; \{C\} \; \textbf{else} \; \{C\} \mid x := e \mid \textbf{skip} \mid \textbf{custom } C'$$

Fig. 2. Syntax of statements in CoreIVL. A is an assertion, x a variable, b a Boolean expression, e an arbitrary expression. Assertions and expressions are represented semantically as sets of states and partial functions from states to values, respectively. C' represents custom statements and is a parameter of the language.

Manipulating SL states via inhale and exhale. At the core of these verifiers is the SL state they track throughout the verification, typically containing a heap (a mapping from heap locations to values) and SL resources (such as fractional permissions to heap locations). This SL state is manipulated with two verification primitives: **inhale** A (also called *assume** and *produce*) and **exhale** A (also called *assert** and *consume*), where A is a separation logic assertion. **inhale** A assumes the logical constraints in A (e.g., constraints on integer values), and adds the resources (e.g., ownership of heap locations) specified by A to the current state. Dually, **exhale** A asserts that the logical constraints in A hold, and removes the resources specified by A from the current state. These two primitives can encode the verification conditions for a wide variety of program constructs. For instance, a procedure call is encoded as exhaling the call's precondition (to check its logical constraints and transfer ownership of resources from caller to callee), followed by inhaling the postcondition (to assume logical constraints and gain resources back from the call).

Diversity of logics and their semantics. While SL-based IVLs all employ some version of these two inhale and exhale primitives, their actual logics are surprisingly diverse in both core connectives and their semantics. GIL and VeriFast support different separation logics, while Viper uses implicit dynamic frames (IDF), a variation of separation logic that allows for heap-dependent expressions in assertions (e.g., separation logic's points-to predicate $e.f \mapsto v$ is expressed as $\textbf{acc}(e.f) * e.f = v$ in IDF, in which the ownership of the heap location and a logical constraint on its value are expressed as two separate conjuncts)[2].

IVLs also support different SL connectives: Viper supports iterated separating conjunctions [Müller et al. 2016a], Viper and Gillian support magic wands [Dardinier et al. 2022b; Schwerhoff and Summers 2015], Viper and VeriFast support fractional recursively-defined predicates [Boyland 2003; Dardinier et al. 2022a], and VeriFast supports arbitrary existential quantification.

A standard approach for generic reasoning over large classes of separation logics is to build reasoning principles based on a *separation algebra* (built over a partial commutative monoid) [Calcagno et al. 2007; Dockins et al. 2009]. We extend this classic concept to a novel notion of *IDF algebra*, which can model separation logics and IDF alike. In particular, IDF algebras allow asserting knowledge about the value of heap locations $e.f$ without asserting ownership of the heap location itself.

Core Language. The syntax of CoreIVL is shown in Fig. 2. To capture the diversity of assertions supported in existing SL-based IVLs, assertions A in our core language are *semantic*, *i.e.*, assertions are *sets of states* (as opposed to fixing a syntax, and having the semantics for this syntax determine the set of states in which a syntactic assertion is true); states themselves are taken from any chosen IDF algebra. Similarly, expressions e are semantically represented as partial functions from states to values. Moreover, although we assume some core statements in our language, we allow these to be arbitrarily extended via a parameter for *custom statements* C', for instance, to add field assignments. The statements of our core language contain the key verification primitives **inhale** and **exhale** described above, as well as **havoc**, which non-deterministically assigns a value to a

---

[2]This difference also affects the semantic models; separation logic is typically formalized using partial heaps, whereas IDF typically uses a total heaps model [Parkinson and Summers 2012].

```
method main(p: Cell)              method main_ivl(p: Ref) {
  // requires acc(p.v, _)           inhale acc(p.v, _)                method l(p,q:Ref){
{                                                                      inhale P_l
  q := alloc(0)                     havoc q                            q.v := p.v
                                    inhale acc(q.v) * q.v = 0          exhale Q_l
                                                                     }
  // {P_l}           {P_r}          exhale P_l * P_r
  q.v := p.v || tmp := p.v          havoc tmp
  // {Q_l}           {Q_r}          inhale Q_l * Q_r                  method r(p,q:Ref){
                                                                       inhale P_r
  tmp := tmp + q.v                  tmp := tmp + q.v                   tmp := p.v
                                                                       exhale Q_r
  free(q)                           exhale acc(q.v)                  }

  assert tmp = p.v + p.v            exhale tmp = p.v + p.v
}                                 }
```

Fig. 3. A simple parallel program (left), annotated with a method precondition, as well as pre- and postconditions for the parallel branches, and its encoding into CoreIVL (instantiated to model Viper), consisting of a main IVL method (middle) and two further methods (right) modeling the parallel branches (that is, the premises of CSL's parallel composition rule). We use the shorthands $P_l \triangleq \mathbf{acc}(p.v, \_) * \mathbf{acc}(q.v)$, $Q_l \triangleq \mathbf{acc}(p.v, \_) * \mathbf{acc}(q.v) * p.v = q.v$, $P_r \triangleq \mathbf{acc}(p.v, \_)$, and $Q_r \triangleq \mathbf{acc}(p.v, \_) * tmp = p.v$, where the IDF assertion $\mathbf{acc}(e, \_)$ expresses non-zero permission to e (corresponding to the SL assertion $\exists p, v. e \xmapsto{p} v$).

variable. Combined with conditional branching, **inhale**, **exhale**, and **havoc** allow us to encode many important statements, such as while loops, procedure calls, and even proof rules for parallel programs, as we show in the next subsection.

2.2 Background: Translational Verification of a Parallel Program

We use the parallel program on the left in Fig. 3 to illustrate how translational verification works, and the challenges that arise in formalizing this widely-used approach. This program takes as input a Cell p (an object with a value field v), allocates a new Cell q, assigns the value of p.v in parallel to q.v and to the variable tmp, then adds the value of q.v to tmp, deallocates q, and finally asserts that tmp is equal to p.v + p.v. Our goal is to verify this program in Concurrent Separation Logic (CSL) [O'Hearn 2004], that is, by encoding the program and the proof rules of CSL into CoreIVL. In particular, we want to prove that the assertion on its last line holds.

Although the original CSL is presented via standard separation logic syntax, we use the syntax of IDF to annotate this example. The syntax $\mathbf{acc}(e.v, f)$ denotes *fractional permission* (ownership) of the heap location $e.v$ (where $f = 1$ allows reading and writing, and a fraction $0 < f < 1$ allows reading) [Boyland 2003]. The syntax $\mathbf{acc}(p.v, \_)$ (used as precondition in our example) denotes a so-called *wildcard permission* (or *wildcard* in short); it is shorthand for $\exists f > 0. \mathbf{acc}(p.v, f)$, which guarantees read access while abstracting the precise fraction.

Correctness of our example means proving a CSL triple $\Delta \vdash_{CSL} [\mathbf{acc}(p.v, \_)] \; C \; [\top]$, where C is the body of the method main in the front-end (left) program ($\top$ is the trivial postcondition). Instead of constructing a proof directly, a translational verifier maps this to an IVL program (shown as a CoreIVL program to the middle and right of Fig. 3) whose correctness implies the existence of a CSL proof for the original program.

Encoding the program into CoreIVL. Our encoding models each proof task of the CSL verification problem as a separate IVL method, whose statements reflect the individual proof steps [Leino and Müller 2009]. The IVL methods main_ivl, l and r are constructed such that the correctness of *all three* implies the existence of a valid CSL proof for main.

The precondition **acc**(p.v, _) of main is modeled by the first **inhale** statement in main_ivl, reflecting that the proof of the main method may rely on the resources and assumptions guaranteed by this precondition. The allocation q := **alloc**(0) is then encoded via a **havoc** and an **inhale** statement to non-deterministically choose a memory location and obtain a full (*i.e.*, 1) permission. Dually, the deallocation **free**(q) after the parallel composition is encoded via an **exhale** statement, which removes this (full) permission from the IVL state. Since permissions are non-duplicable (technically, affine) resources, this encoding guarantees that no permission can remain and so any attempt to later access this location would cause a verification failure.

To understand the encoding of a source-level parallel composition, we recall the CSL proof rule[3]:

$$\text{PAR} \; \frac{\Delta \vdash_{CSL} [P_l] \; C_l \; [Q_l] \qquad \Delta \vdash_{CSL} [P_r] \; C_r \; [Q_r]}{\Delta \vdash_{CSL} [P_l * P_r] \; C_l \; || \; C_r \; [Q_l * Q_r]}$$

From the point of view of the *outer thread* (forking and joining the parallel branches), the overall effect of the parallel composition can be seen as *giving up* the separating conjunction $P_l * P_r$ of the preconditions of the parallel branches, and obtaining the corresponding postconditions $Q_l * Q_r$ before resuming any remaining code[4]. This exchange of assertions across the triple in the conclusion of the rule (as well as the intervening modification of tmp) is modeled in the IVL program by the sequence **exhale** $P_l * P_r$; **havoc** tmp; **inhale** $Q_l * Q_r$.

The premises of the parallel rule are checked by verifying two extra methods l and r, whose pre- and postconditions correspond to the Hoare triples from the rule premises directly. The encoded bodies of l and r follow the standard pattern: an **inhale** of their preconditions (which can be seen as the other "half" of the transfer from the outer thread, modeled by **exhale** $P_l * P_r$), the translation of their source implementations, and finally an **exhale** of their postconditions.

If running a back-end verifier for the IVL on the three encoded methods succeeds, we have demonstrated that a CSL proof for the original program exists—provided that the translational verification is sound. Soundness depends on a non-trivial translation, the subtle semantics of an IVL, and the algorithms employed by back-end verifiers. In the rest of this section, we explain our formal framework for establishing the soundness of translational verifiers.

2.3 Operational Semantics and Back-End Verifiers

To make formal claims about an IVL program, we need a formal semantics and notion of correctness for the IVL itself. As explained in the introduction, an operational semantics facilitates a formal connection to various back-end algorithms, which typically have an operational flavor. Since our semantics needs to capture verification algorithms that make heavy use of (demonic) non-determinism (to model concurrency, allocation, or abstract modularly over the precise behavior of program elements), our operational semantics embraces such non-determinism. Moreover, to account for the diversity of the verification algorithms used in back-ends, our semantics also incorporates the dual notion of *angelic non-determinism*.

Consider verifying the statement **exhale acc**(a.v) ∨ **acc**(b.v), which requires giving up (full) permission to *either* a.v or b.v; if the original state holds both permissions, either choice avoids

---

[3]We omit technical side-conditions from the original rule that restrict mutation of variables shared amongst threads; these are taken care of properly in real verifiers and our formalizations.

[4]We assume (as is common for modular verifiers) that each thread's specification is explicitly annotated, as in main.

$$\text{InhaleOp} \quad \frac{}{\langle \mathbf{inhale}\ A, \omega \rangle \rightarrow_\Delta \{\omega\} * A}$$

$$\text{ExhaleOp} \quad \frac{\omega = \omega' \oplus \omega_A \qquad \omega_A \in A}{\langle \mathbf{exhale}\ A, \omega \rangle \rightarrow_\Delta \{\omega'\}}$$

$$\text{SeqOp} \quad \frac{\langle C_1, \omega \rangle \rightarrow_\Delta S_1 \qquad \forall \omega_1 \in S_1.\ \langle C_2, \omega_1 \rangle \rightarrow_\Delta \mathcal{S}(\omega_1)}{\langle C_1; C_2, \omega \rangle \rightarrow_\Delta \cup_{\omega_1 \in S_1} \mathcal{S}(\omega_1)}$$

(a) Selected operational semantics rules.

$$\text{InhaleAx} \quad \frac{}{\Delta \vdash [P]\ \mathbf{inhale}\ A\ [P * A]}$$

$$\text{ExhaleAx} \quad \frac{P \models Q * A}{\Delta \vdash [P]\ \mathbf{exhale}\ A\ [Q]}$$

$$\text{SeqAx} \quad \frac{\Delta \vdash [P]\ C_1\ [R] \qquad \Delta \vdash [R]\ C_2\ [Q]}{\Delta \vdash [P]\ C_1; C_2\ [Q]}$$

(b) Selected axiomatic semantic rules.

Fig. 4. Selected simplified operational and axiomatic semantic rules.

a failure here, but results in different successor states, and so might affect whether subsequent statements verify successfully. Such algorithmic choices occur for other IVL constructs, such as for choosing the values of existentials (including the amount of permission for a wildcard permission), or determining the footprints of magic wands. Our operational semantics makes *all algorithmic choices possible* and defines a program as correct if *any* such choice avoids failure.

Operational semantics. To capture the dual non-determinism, we define our operational semantics as a multi-relation [Rewitzky 2003; Guéneau et al. 2023]

$$\langle C, \omega \rangle \rightarrow_\Delta S$$

where C is an IVL statement, ω an initial state, S a set of final states, and Δ a type context (mapping for example variables to types, *i.e.*, to sets of values). The set S captures the *demonic choices*, *i.e.*, contains the resulting state for each possible demonic choice. On the other hand, *angelic choices* are reflected by *different result sets* derivable in our semantics. Returning to our previous example, if ω is a state with full permission to both a.v and b.v, our semantics allows for both transitions $\langle \mathbf{exhale}\ \mathbf{acc}(\mathtt{a.v}) \lor \mathbf{acc}(\mathtt{b.v}), \omega \rangle \rightarrow_\Delta \{\omega_{-a}\}$ and $\langle \mathbf{exhale}\ \mathbf{acc}(\mathtt{a.v}) \lor \mathbf{acc}(\mathtt{b.v}), \omega \rangle \rightarrow_\Delta \{\omega_{-b}\}$ (where ω_{-a} and ω_{-b} are identical to ω but with the permission to a.v resp. b.v removed).

A successful verification by a back-end is represented by an execution in our operational semantics, leading to the following definition of correctness of a CoreIVL statement:

Definition 1. A CoreIVL statement C is *correct* for a well-formed initial state ω iff C executes successfully in ω, *i.e.*, $\exists S.\ \langle C, \omega \rangle \rightarrow_\Delta S$. C is *valid* iff it is correct for all well-formed initial states.

Fig. 4a shows simplified rules for the operational semantics of **inhale** A, **exhale** A, and sequential composition. The (non-simplified) rules for all statements are shown in §3. Inhaling A in state ω leads to the set of all possible combinations $\omega \oplus \omega_A$ for $\omega_A \in A$, capturing the demonic non-determinism of **inhale**: All possible states satisfying A must be considered in the rest of the program. Dually, the rule ExhaleOp allows *any choice of state* ω_A satisfying A (that is, uses angelic non-determinism), and to remove it from ω. In our previous example, ω can be decomposed into $\omega = \omega_{-a} \oplus \omega_a$ or $\omega = \omega_{-b} \oplus \omega_b$, where ω_a and ω_b respectively contain the permission to a.v and b.v (and thus ω_a and ω_b both satisfy the exhaled assertion $\mathbf{acc}(\mathtt{a.v}) \lor \mathbf{acc}(\mathtt{b.v})$). The rule SeqOp for sequential composition is more involved, since it needs to deal with the dual non-determinism: It requires a single function $\mathcal{S}$ that maps every state ω_1 from S_1 (the set of states obtained after executing C_1 in ω) to a set of states $\mathcal{S}(\omega_1)$ that can be reached by executing C_2 in ω_1. The choice of the function $\mathcal{S}$ captures the angelic choices in C_2.

Connection to back-end verifiers. To show that this operational semantics for CoreIVL is indeed suitable to capture different verification algorithms, we connect it to formalizations of the

two main back-ends used by Viper. First, we formalize a version of Viper's symbolic execution back-end [Schwerhoff 2016] in Isabelle/HOL and prove it sound against the operational semantics of CoreIVL. Second, we connect the formalization of Viper's verification condition generation back-end by Parthasarathy et al. [2024] to CoreIVL by constructing a CoreIVL execution from a successful verification by their back-end. The soundness proofs of these back-ends are described in §4. There we will also see that the angelic choice described earlier in this section is crucial for enabling these proofs since the two back-ends use different heuristics, in particular around exhaling wildcard permissions.

2.4 Axiomatic Semantics

The previously-introduced definition of correctness (Def. 1) based on the operational semantics is well-suited to connect to back-end verifiers. However, connecting it to front-end programs, and especially logics such as CSL in our example from Fig. 3, requires substantial effort due to the large semantic gap between the operational IVL semantics and the front-end logic. The IVL semantics presented previously is *operational*, describes the execution from a *single state*, and exposes low-level details (such as handling the dual non-determinism in the rule SᴇqOᴘ). In contrast, the program logic is *axiomatic*, describes the behavior of *sets of states* (via assertions), and is more high-level (*e.g.*, it uses an intermediate assertion in the rule SᴇqAx instead of the semantic function $\mathcal{S}$). To bridge this gap, we present an alternative (and, as we later prove, equivalent) axiomatic semantics for CoreIVL, which is closer to the separation logics typically used for front-end programs and, thus, simplify the proof that a front-end translation is sound.

Our axiomatic semantics uses judgments of the form

$$\Delta \vdash [P] \; C \; [Q]$$

where P and Q are semantic assertions (sets of states), C is an IVL statement, and Δ is a type context. Intuitively, this triple expresses that C can be executed successfully in any state from P (with the right angelic choices), and Q is (precisely) the set of all states reached by these executions. Formally, we want the following *soundness* property (we will present the completeness theorem in §3):

Theorem 2 (Operational-to-Axiomatic Soundness.)**.** *If the CoreIVL statement C is well-typed and valid (Def. 1) then there exists a set of states B such that $\Delta \vdash [\top] \; C \; [B]$ holds.*

Note that, in contrast to when one defines a proof system for a pre-existing operational semantics, the desired implication here is from operational to axiomatic semantics; this is due to the connection we are aiming for from back-end algorithms (defined operationally) to front-end proofs.

The rules for the axiomatic semantics of **inhale** A, **exhale** A, and sequential composition are shown in Fig. 4b. The rule InhaleAx for **inhale** A corresponds to the operational rule InhaleOp, where ω has been lifted to the set of states P (since $P * A = \bigcup_{\omega \in P}(\{\omega\} * A)$). The rule ExhaleAx for **exhale** A is more involved, as it first requires weakening the set of initial states P to $Q * A$. Weakening is in general necessary to disentangle the states in Q and A: For example, to exhale **acc**(a.v) from a precondition **acc**(a.v) $*$ **acc**(b.v) $*$ a.v = b.v, we have to first drop the equality a.v = b.v because otherwise the resulting postcondition would refer to a memory location that is no longer owned. Moreover, similarly to how Hoare logic hides the induction necessary to reason about unbounded while loops behind a loop invariant, our axiomatic semantics hides the dual non-determinism of the operational semantics behind high-level connectives such as the separating conjunction. Intuitively, in the rule ExhaleAx, the angelic choice is hidden in the choice of Q and the split of every state in P into a state in Q and a state in A. In our previous example **exhale acc**(a.v) $\vee$ **acc**(b.v), we could choose Q to be either **acc**(a.v) or **acc**(b.v), *i.e.*, we

FRAME
$$\frac{\Delta \vdash_{CSL} [P]\ C\ [Q] \qquad fv(F) \cap mod(C) = \emptyset}{\Delta \vdash_{CSL} [P * F]\ C\ [Q * F]}$$

PAR
$$\frac{\Delta \vdash_{CSL} [P_l]\ C_l\ [Q_l] \qquad \Delta \vdash_{CSL} [P_r]\ C_r\ [Q_r] \qquad \dots}{\Delta \vdash_{CSL} [P_l * P_r]\ C_l\ ||\ C_r\ [Q_l * Q_r]}$$

SEQ
$$\frac{\Delta \vdash_{CSL} [P]\ C_1\ [R] \qquad \Delta \vdash_{CSL} [R]\ C_2\ [Q]}{\Delta \vdash_{CSL} [P]\ C_1; C_2\ [Q]}$$

CONS
$$\frac{\Delta \vdash_{CSL} [P']\ C\ [Q'] \qquad P \models P' \qquad Q' \models Q}{\Delta \vdash_{CSL} [P]\ C\ [Q]}$$

ALLOC
$$\frac{r \notin fv(e)}{\Delta \vdash_{CSL} [\top]\ \mathsf{r} := \mathbf{alloc}(e)\ [\mathbf{acc}(\mathsf{r.v}) * r.v = e]}$$

FREE
$$\Delta \vdash_{CSL} [\mathbf{acc}(\mathsf{q.v})]\ \mathbf{free}(\mathsf{q})\ [\top]$$

Fig. 5. Selected CSL rules. In the rule FRAME, $fv(F)$ and $mod(C)$ denote the set of variables free in F and the set of variables potentially modified by C, respectively.

could derive both $\Delta \vdash [\mathbf{acc}(\mathsf{a.v}) * \mathbf{acc}(\mathsf{b.v})]\ \mathbf{exhale}\ \mathbf{acc}(\mathsf{a.v}) \vee \mathbf{acc}(\mathsf{b.v})\ [\mathbf{acc}(\mathsf{a.v})]$ and $\Delta \vdash [\mathbf{acc}(\mathsf{a.v}) * \mathbf{acc}(\mathsf{b.v})]\ \mathbf{exhale}\ \mathbf{acc}(\mathsf{a.v}) \vee \mathbf{acc}(\mathsf{b.v})\ [\mathbf{acc}(\mathsf{b.v})]$.

Finally, the rule SEQAX for sequential composition illustrates how the axiomatic semantics abstracts over the low-level details of the dual non-determinism in the operational semantics, such as the existence of the semantic function $\mathcal{S}$ in rule SEQOP. Instead, the axiomatic rule SEQAX uses an intermediate assertion R; its relation to $\mathcal{S}$ is proved once and for all in the soundness proof and, thus, does not have to be proved for each front-end.

Crucially, we have designed the axiomatic semantics such that it contains *exactly one rule* per statement. In particular, it contains no structural rules such as a frame rule or a consequence rule, which are not necessary in our setting. This allows us to deconstruct an axiomatic semantic derivation into smaller blocks, to then reconstruct a proof in the front-end logic. For example, one can derive from $\Delta \vdash [P]\ C_1; C_2\ [Q]$ the existence of some assertion R such that $\Delta \vdash [P]\ C_1\ [R]$ and $\Delta \vdash [R]\ C_2\ [Q]$ hold. Using this axiomatic semantics, we can now easily connect the correctness of the IVL program to the correctness of the front-end program, as we explain next.

Connecting to front-end programs and logics. Let us now see how the axiomatic semantics enables us to construct a CSL proof for the front-end program from Fig. 3. Concretely, we build a CSL proof of the triple $\Delta \vdash_{CSL} [\mathbf{acc}(\mathsf{p.v}, \_)]\ C\ [\top]$, where C corresponds to the body of the method main. To do this, we use the CSL rules shown in Fig. 5 and the CoreIVL triples $\Delta \vdash [\top]\ C\ [B]$ for the methods l, r, and main_ivl that we obtain from Thm. 2.

The first step of proving the CSL triple for main is to pair each statement in main with the corresponding code in main_ivl. For this, we use CSL's SEQ rule and (the inversion of) SEQAX to split the proofs for main and main_ivl into smaller parts:

$\Delta \vdash [\top]\ \mathbf{inhale}\ \mathbf{acc}(\mathsf{p.v}, \_)\ [A_0]$

$\Delta \vdash_{CSL} [A_0]\ \mathsf{q} := \mathbf{alloc}(0)\ [A_1]$ $\Delta \vdash [A_0]\ \mathbf{havoc}\ \mathsf{q}; \mathbf{inhale}\ \mathbf{acc}(\mathsf{q.v}) * \mathsf{q.v} = 0\ [A_1]$

$\Delta \vdash_{CSL} [A_1]\ \mathsf{q.v} := \mathsf{p.v}\ ||\ \mathsf{tmp} := \mathsf{p.v}\ [A_2]$ $\Delta \vdash [A_1]\ \mathbf{exhale}\ P_l * P_r; \mathbf{havoc}\ \mathsf{tmp}; \mathbf{inhale}\ Q_l * Q_r\ [A_2]$

$\Delta \vdash_{CSL} [A_2]\ \mathsf{tmp} := \mathsf{tmp} + \mathsf{q.v}\ [A_3]$ $\Delta \vdash [A_2]\ \mathsf{tmp} := \mathsf{tmp} + \mathsf{q.v}\ [A_3]$

$\Delta \vdash_{CSL} [A_3]\ \mathbf{free}(\mathsf{q})\ [A_4]$ $\Delta \vdash [A_3]\ \mathbf{exhale}\ \mathbf{acc}(\mathsf{q.v})\ [A_4]$

$\Delta \vdash_{CSL} [A_4]\ \mathbf{assert}\ \mathsf{tmp} = \mathsf{p.v} + \mathsf{p.v}\ [B]$ $\Delta \vdash [A_4]\ \mathbf{exhale}\ \mathsf{tmp} = \mathsf{p.v} + \mathsf{p.v}\ [B]$

Note how deconstructing the applications of SEQAx in the proof of main_ivl gives us intermediate assertions A_{0-4}, which we use to instantiate the intermediate assertion R in SEQ.[5] Matching statements of the front-end program to segments of the CoreIVL program is straightforward since the front-end translation is typically defined statement by statement.

After decomposing the sequential compositions, we justify the CSL triple for each primitive front-end statement from the corresponding CoreIVL triple. For some statements like tmp := tmp + q.v, this is trivial as the triples (and corresponding logic rules) match. Let us now focus on the most interesting cases: q := **alloc**(0), q.v := p.v || tmp := p.v, and **free**(q).

The exhale-havoc-inhale pattern. To derive the CSL triples for these statements, we observe that their encoding follows a pattern: The CoreIVL code first exhales the precondition P of the CSL rule (omitted if $P = \top$), then havocs the variables modified by the statement (q for q := **alloc**(0) and tmp for q.v := p.v || tmp := p.v), and finally inhales the postcondition Q of the CSL rules (omitted if $Q = \top$), leading to the pattern **exhale** P; **havoc** x_1; . . . ; **havoc** x_n; **inhale** Q. To handle this general pattern, we can use the following lemma, which holds for any separation logic $\mathcal{L}$ with a consequence rule and a frame rule (see §5 for the proof):

Lemma 1 (Exhale-havoc-inhale). *For any separation logic $\mathcal{L}$ that has a frame rule and a consequence rule, if $\Delta \vdash_{\mathcal{L}} [P] \, C \, [Q]$ holds and $\Delta \vdash [A]$ **exhale** P; **havoc** x_1; . . . ; **havoc** x_n; **inhale** Q $[B]$ holds, where $\{x_1, \ldots, x_n\} = mod(C)$, then $\Delta \vdash_{\mathcal{L}} [A] \, C \, [B]$ holds.*

Intuitively, this lemma shows that a CoreIVL triple for the exhale-havoc-inhale pattern allows us to obtain the corresponding CSL triple. In the case of q := **alloc**(0), this lets us lift ALLOC to the precondition A_0 and postcondition A_1, giving us exactly the triple we need. To justify the triple for q.v := p.v || tmp := p.v, we need to establish the premises of the rule PAR, $\Delta \vdash_{CSL} [P_l] \, q.v := p.v \, [Q_l]$ and $\Delta \vdash_{CSL} [P_r] \, tmp := p.v \, [Q_r]$, which can be derived from the correctness of the methods l and r using a lemma similar to Lemma 1, as we formally show in §5.

Summary. We have now seen how to justify the translational verification of the program from Fig. 3 in CSL in three steps. First, we showed that the successful verification of its CoreIVL encoding in a back-end implies that the CoreIVL program is valid. Second, we used the soundness theorem for the axiomatic IVL semantics to derive judgments in the axiomatic semantics. Third, we use those judgments to prove the desired CSL triple. Each of these steps is well-suited for its task: The operational semantics allows us to connect to the back-end verifiers, while the axiomatic semantics facilitates the reconstruction of the front-end logic proof—both linked by Thm. 2.

3　Semantics

In this section, we present an operational and an axiomatic semantics for the CoreIVL language defined in Fig. 2. We first define in §3.1 an IDF algebra that captures both separation logic and implicit dynamic frames state models. We then formalize the operational semantics of CoreIVL in §3.2 and define its axiomatic semantics and prove their equivalence in §3.3. We instantiate CoreIVL for key features of Viper in §3.4.

3.1　An Algebra for Separation Logic and Implicit Dynamic Frames

A standard way to capture different separation logic state models is to use a *separation algebra* [Calcagno et al. 2007; Dockins et al. 2009], *i.e.*, a partial commutative monoid $(\Sigma, \oplus)$, where Σ is the set of all states, and $\oplus$ is a partial, commutative, and associative binary operator, used to

---

[5]Note that the CSL we use in this paper has the same state model as the IVL, and thus the IVL assertions do not need to be converted to CSL assertions. Our axiomatic semantics can also be used to reconstruct proofs in program logics with different state models, but this goes beyond the scope of this paper.

$$a \oplus b = b \oplus a \qquad a \oplus (b \oplus c) = (a \oplus b) \oplus c \qquad c = a \oplus b \wedge c = c \oplus c \Rightarrow a = a \oplus a$$

$$x = x \oplus |x| \qquad |x| = |x| \oplus |x| \qquad x = x \oplus c \Rightarrow |x| \succeq c \qquad |a \oplus b| = |a| \oplus |b|$$

$$x \oplus a = x \oplus b \wedge |a| = |b| \Rightarrow a = b \qquad \text{stable}(\omega) \Rightarrow \omega = \text{stabilize}(\omega) \qquad \text{stable}(\text{stabilize}(\omega))$$

$$\text{stabilize}(a \oplus b) = \text{stabilize}(a) \oplus \text{stabilize}(b) \qquad x = \text{stabilize}(x) \oplus |x| \qquad a = b \oplus \text{stabilize}(|c|) \Rightarrow a = b$$

Fig. 6. Axioms for our IDF algebra $(\Sigma, \oplus, |\_|, \text{stable}, \text{stabilize})$. We define $(\omega' \succeq \omega) \triangleq (\exists r. \omega' = \omega \oplus r)$.

combine states (*e.g.*, via the separating conjunction operator $*$). In SL, assertions about values of heap locations must also assert ownership of those heap locations. In particular, asserting that a heap location $\mathtt{x.f}$ has a value 5 requires using the points-to predicate $\mathtt{x.f} \mapsto 5$), which also expresses ownership of the location $\mathtt{x.f}$. This requirement is embedded in the SL state model. For example, a typical SL state with a heap and fractional permissions (ignoring local variables for now) is $\Sigma_{SL} \triangleq (L \rightharpoonup (V \times (0,1]))$, *i.e.*, a partial function from a set L of heap locations to pairs of values from a set V and positive fractional permissions. That is, any value for a heap location is associated with a strictly positive permission.

In contrast, in implicit dynamic frames, an assertion may constrain the value of a heap location independently of expressing ownership. For example, $\mathtt{x.f} = 5$ is a valid IDF assertion that expresses that $\mathtt{x.f}$ stores the value 5 without expressing ownership of $\mathtt{x.f}$. However, IDF requires assertions used as pre- and postconditions, loop invariants, frames (for the frame rule), etc. to be *self-framing*, that is, to express ownership of all heap locations they mention. For example, $\mathbf{acc}(\mathtt{x.f}) * \mathtt{x.f} = 5$ is self-framing, while $\mathtt{x.f} = 5$ is not. To capture IDF states with fractional permissions, we define the state model $\Sigma_{IDF} \triangleq (L \rightharpoonup V) \times (L \rightharpoonup [0,1])$.[6] In contrast to Σ_{SL}, values and permissions are separated in Σ_{IDF}, which allows states (h, π) where $h(\mathtt{x.f}) = 5$ but $\pi(\mathtt{x.f}) = 0$.

We call a state $(h, \pi) \in \Sigma_{IDF}$ *stable* iff it contains values exactly for the heap locations with non-zero permission, *i.e.*, $\text{dom}(h) = \{l \mid \pi(l) > 0\}$. Stable states are exactly those that can be represented as states in Σ_{SL}; By construction, all states in Σ_{SL} are stable.

To capture arbitrary SL and IDF states, we define an *IDF algebra* as follows:

Definition 3. An *IDF algebra* is a tuple $(\Sigma, \oplus, |\_|, \text{stable}, \text{stabilize})$ that satisfies all axioms in Fig. 6, where Σ is a set of states, $\oplus$ is a partial, commutative, and associative addition on Σ (*i.e.*, a partial function from $\Sigma \times \Sigma$ to Σ), $|\_|$ and stabilize are endomorphisms of Σ, and stable is a predicate on Σ.

The set Σ and the partial addition $\oplus$ are the standard components of a separation algebra. Using $\oplus$, we define the standard partial order $\succeq$ induced by $\oplus$ as $(\omega' \succeq \omega) \triangleq (\exists r. \omega' = \omega \oplus r)$. We require *positivity* ($c = a \oplus b = c \wedge c = c \oplus c \Rightarrow a = a \oplus a$) to ensure that the partial order is antisymmetric ($a \succeq b \wedge b \succeq a \Rightarrow a = b$). Intuitively, the endomorphism $|\_|$ projects a state ω on its largest *duplicable* part, *i.e.*, $|\omega|$ is the largest state smaller than ω such that $|\omega| = |\omega| \oplus |\omega|$. Similarly, the endomorphism stabilize projects a state ω on its largest *stable* part, *i.e.*, $\text{stabilize}(\omega)$ is the largest stable state smaller than ω.

Instantiations. For our concrete IDF state model Σ_{IDF}, the combination $(h_1, \pi_1) \oplus (h_2, \pi_2)$ is defined iff h_1 and h_2 agree on the locations to which both states hold non-zero permission and the sums of their permissions pointwise is at most 1, *i.e.*, iff $\forall l. (\pi_1(l) + \pi_2(l) \leq 1) \wedge (l \in \text{dom}(h_1) \cap \text{dom}(h_2) \Rightarrow h_1(l) = h_2(l))$. When the combination is defined, $(h_1, \pi_1) \oplus (h_2, \pi_2) \triangleq (h_1 \cup h_2, \pi_1 + \pi_2)$. Knowledge about heap values is duplicable, whereas permissions are not. Thus, $|\_|$

---

[6]In addition, values must exist for those heap locations where the state has non-zero permission. That is, Σ_{IDF} is restricted to states (h, π) such that $\forall l. \pi(l) > 0 \Rightarrow l \in \text{dom}(h)$.

puts all permissions to 0 but preserves the heap, *i.e.*, $|(h, \pi)| \triangleq (h, \lambda l. 0)$. Moreover, stabilize erases all values for heap locations to which the state does not hold any permission, *i.e.*, $\text{stabilize}((h, \pi)) \triangleq ((\lambda l. \text{ if } \pi(l) > 0 \text{ then } h(l) \text{ else } \bot), \pi)$.

Separation algebra instances can also be instantiated as IDF algebras, by defining stable to be true for all states, and stabilize to be the identity function on Σ. For example, Σ_{SL} (defined above) can be instantiated as an IDF algebra with these definitions of stable and stabilize, and with $|\_|$ mapping every state to the unit state (where all permissions are 0, and the domain of the heap is empty). Moreover, like separation algebras [Dockins et al. 2009; Jung et al. 2015], IDF algebras support standard constructions like the agreement algebra (where only $\omega = \omega \oplus \omega$ holds), and can be constructed by combining smaller algebras, via combinators such as product and sum types (where both types must be IDF algebras), function types (where only the codomain must be an IDF algebra), etc.

State model for CoreIVL. Our CoreIVL framework can be instantiated for any IDF algebra. We obtain the state model by extending this IDF algebra with a store of local variables, *i.e.*, a partial mapping from variables in *Var* to values in *Val*. Concretely, given an IDF algebra with carrier set Σ, we define the state model for CoreIVL as the product algebra $\Sigma_{IVL} \triangleq ((Var \rightharpoonup Val) \times \Sigma)$, where the store $Var \rightharpoonup Val$ is instantiated to the agreement algebra, *i.e.*, addition on stores is defined for identical stores (as the identity). Using the agreement algebra for the store ensures that **inhale** and **exhale** have no effect on local variables.

Self-framing IDF assertions. Given an arbitrary IDF algebra, we can define a general notion of *self-framing* assertions and assertions *framing* other assertions as follows.

Definition 4. In the following, *P* is an IDF assertion (*i.e.*, a set of states from an IDF algebra).

- *P* is *self-framing*, written $\text{selfFraming}(P)$, iff $\forall \omega. \omega \in P \Leftrightarrow \text{stabilize}(\omega) \in P$.
- A state ω *frames* *P*, written $\text{frames}(\omega, P)$, iff $\text{selfFraming}(\{\omega\} * P)$.
- An IDF assertion *B* *frames* *P*, written $\text{frames}(B, P)$, iff $\forall \omega \in B. \text{stable}(\omega) \Rightarrow \text{frames}(\omega, P)$.
- *P* *frames* an *expression* (*i.e.*, a partial function from states to values) *e*, written $\text{frames}(P, e)$, iff $e(\omega)$ is defined for all states $\omega \in P$.

Those different notions are tightly connected: If *A* is self-framing and *A* frames *B* then $A * B$ is self-framing. For example, the assertion $A \triangleq (\textbf{acc}(\texttt{x.f}) * \texttt{x.f} = 5)$ is self-framing, because any state $\omega_A \in A$ has full permission to $\texttt{x.f}$, and thus $\text{stabilize}(\omega_A)$ will retain the knowledge that $\texttt{x.f}$ is 5, and hence $\text{stabilize}(\omega_A) \in A$. In contrast, the assertion $B \triangleq (\texttt{x.f} = 5)$ is not self-framing, since a state ω_B with no permission to $\texttt{x.f}$ but with the knowledge that $\texttt{x.f}$ is 5 satisfies *B*, but $\text{stabilize}(\omega_B)$ will not retain the knowledge that $\texttt{x.f} = 5$, and hence will not satisfy *B*. Moreover, any state that satisfies $\textbf{acc}(\texttt{x.f})$ frames *B*, thus the assertion $\textbf{acc}(\texttt{x.f})$ frames *B*. Note that, in an instantiation with SL states (*e.g.*, Σ_{SL}), all assertions are self-framing, since all SL states are stable.

3.2 Operational Semantics

We now formally define the operational semantics of CoreIVL for the state model described above (given an arbitrary IDF algebra). As explained in §2.3, our operational semantics has judgments of the form $\langle C, \omega \rangle \rightarrow_\Delta S$, where Δ is a type context,[7] *C* is a statement, ω is a state, and *S* is a set of states (to capture demonic non-determinism; angelic non-determinism is captured by the existence of different derivations $\langle C, \omega \rangle \rightarrow_\Delta S_1$ and $\langle C, \omega \rangle \rightarrow_\Delta S_2$).

---

[7]In this section, we do not discuss typing in details, but our Isabelle formalization includes it. In particular, it ensures that our operational and axiomatic semantics deal only with well-typed states, *i.e.*, states whose local store and heap contain values of the right types (defined by the type context Δ). By default, all states discussed in this section are well-typed.

INHALEOP
$$\frac{frames(\omega, A)}{\langle \textbf{inhale } A, \omega \rangle \rightarrow_\Delta \{\omega' \mid \exists \omega_A \in A.\, \omega' = \omega \oplus \omega_A \land stable(\omega')\}}$$

EXHALEOP
$$\frac{\omega = \omega' \oplus \omega_A \qquad \omega_A \in A \qquad stable(\omega')}{\langle \textbf{exhale } A, \omega \rangle \rightarrow_\Delta \{\omega'\}}$$

SEQOP
$$\frac{\langle C_1, \omega \rangle \rightarrow_\Delta S_1 \qquad \forall \omega_1 \in S_1.\, \langle C_2, \omega_1 \rangle \rightarrow_\Delta \mathcal{S}(\omega_1)}{\langle C_1; C_2, \omega \rangle \rightarrow_\Delta \cup_{\omega_1 \in S_1} \mathcal{S}(\omega_1)}$$

ASSIGNOP
$$\frac{\Delta(x) = \tau \qquad e(\omega) = v \qquad v \in \tau}{\langle x := e, \omega \rangle \rightarrow_\Delta \{\omega[x \mapsto v]\}}$$

SKIPOP
$$\frac{}{\langle \textbf{skip}, \omega \rangle \rightarrow_\Delta \{\omega\}}$$

HAVOCOP
$$\frac{\Delta(x) = \tau}{\langle \textbf{havoc } x, \omega \rangle \rightarrow_\Delta \{\omega[x \mapsto v] \mid v \in \tau\}}$$

IFTOP
$$\frac{b(\omega) = \top \qquad \langle C_1, \omega \rangle \rightarrow_\Delta S_1}{\langle \textbf{if}(b) \ \{C_1\} \ \textbf{else} \ \{C_2\}, \omega \rangle \rightarrow_\Delta S_1}$$

IFFOP
$$\frac{b(\omega) = \bot \qquad \langle C_2, \omega \rangle \rightarrow_\Delta S_2}{\langle \textbf{if}(b) \ \{C_1\} \ \textbf{else} \ \{C_2\}, \omega \rangle \rightarrow_\Delta S_2}$$

Fig. 7. Operational semantics rules.

The rules for the operational semantics are given in Fig. 7. As shown by the rule INHALEOP, **inhale** A can reduce in a state ω only if ω frames A. In our concrete instantiation Σ_{IDF}, this means that ω or A must contain the permission to any heap location mentioned in A. For example, **inhale** x.f = 5 can reduce correctly only in a state ω that has some permission to x.f. If ω has a different value than 5 for x.f, the statement will reduce to an empty set of states, *i.e.*, $\langle \textbf{inhale } \text{x.f} = 5, \omega \rangle \rightarrow_\Delta \varnothing$, capturing the fact that we inhaled an assumption inconsistent with our state. In this case, the rest of the program is trivially correct (because it will be executed in no state). If ω has value 5 for x.f, then the statement will reduce to the singleton set $\{\omega\}$, *i.e.*, $\langle \textbf{inhale } \text{x.f} = 5, \omega \rangle \rightarrow_\Delta \{\omega\}$. Finally, inhaling **acc**(x.f) in a state ω with no permission and no value to x.f will result in a set with multiple states (potentially infinitely many), one state for each possible value of x.f. We require $stable(\omega')$ in the rule to ensure that executing a statement in any stable state leads to a set of stable states, *i.e.*, $\forall \omega.\, stable(\omega) \land \langle \omega, C \rangle \rightarrow_\Delta S \Rightarrow (\forall \omega' \in S.\, stable(\omega'))$. In other words, the operational semantics preserves the stability of states.

Dually, the rule EXHALEOP requires the final state ω' to be stable. This ensures that values of heap locations for which the state lost all permission will be erased. For example, **exhale acc**($x.f$) succeeds only in a state with full permission to x.f, and results in a final state without any permission nor value for x.f. Note that the rule EXHALEOP is the only atomic rule that uses angelic nondeterminism, because the rule can be applied with different ω' (corresponding to different angelic choices). (The rules INHALEOP and HAVOCOP use demonic non-determinism, while ASSIGNOP and SKIPOP are deterministic.) The rule SEQOP first executes C_1 in ω, which yields a set of states S_1. Since S_1 captures demonic choices, C_2 must be executed in *all* states from S_1, but the angelism in C_2 can be resolved differently for each state, which is captured by the choice of the function $\mathcal{S}$. The function $\mathcal{S}$ must map every state ω_1 from S_1 (the set of states obtained after executing C_1 in ω) to a set of states $\mathcal{S}(\omega_1)$ that can be reached by executing C_2 in ω_1.

Finally, note that expressions in CoreIVL are *semantic*, *i.e.*, they are *partial* functions from states to values. We model them as partial functions because they might be heap-dependent, and thus might not be defined for all states. For example, the expression x.f = 5 is only meaningful in states where x.f has a value. The rules ASSIGNOP, IFTOP, and IFFOP require that the expressions are defined in the initial state ω.

3.3 Axiomatic Semantics

Using the same extended state model as in the operational semantics, we define an axiomatic semantics with judgments of the form $\Delta \vdash [P] \ C \ [Q]$, where Δ is a type context, P and Q are assertions (sets of states), and C is a CoreIVL statement. All rules are shown in Fig. 8. Multiple rules

SKIPAX
selfFraming(P)

$\Delta \vdash [P]\ \textbf{skip}\ [P]$

INHALEAX
selfFraming(P)　　　frames(A, P)

$\Delta \vdash [P]\ \textbf{inhale}\ A\ [P * A]$

EXHALEAX
selfFraming(P)　　　$P \models Q * A$　　　selfFraming(Q)

$\Delta \vdash [P]\ \textbf{exhale}\ A\ [Q]$

IFAX
selfFraming(P)　　　frames(P, b)　　　$\Delta \vdash [P \wedge b]\ C_1\ [B_1]$　　　$\Delta \vdash [P \wedge \neg b]\ C_2\ [B_2]$

$\Delta \vdash [P]\ \textbf{if}(b)\ \{C_1\}\ \textbf{else}\ \{C_2\}\ [B_1 \vee B_2]$

HAVOCAX
selfFraming(P)　　　$\Delta(x) = \tau$

$\Delta \vdash [P]\ \textbf{havoc}\ x\ [\exists x \in \tau.\,P]$

SEQAX
$\Delta \vdash [P]\ C_1\ [R]$　　　$\Delta \vdash [R]\ C_2\ [Q]$

$\Delta \vdash [P]\ C_1; C_2\ [Q]$

ASSIGNAX
selfFraming(P)　　　frames(P, e)

$\Delta \vdash [P]\ x := e\ [\exists v.\,P[v/x] \wedge x = e[v/x]]$

Fig. 8. Axiomatic semantic rules.

have side-conditions requiring the preconditions and postconditions to be self-framing, ensuring that if we have $\Delta \vdash [P]\ C\ [Q]$, P and Q are self-framing.

As explained in §2.4, our operational and axiomatic semantics are equivalent. The soundness property expressed in Thm. 2 (in §2.4) allows one to bridge the gap between a valid CoreIVL program (according to Def. 1) and the front-end program logic. The proof of Thm. 2 is not straightforward. In particular, our proof explicitly tracks the angelic choices made based on the sequence of past states of each execution, as shown by the following lemma, which implies Thm. 2. Let $\ll A \gg \triangleq \{\omega' \mid \text{stabilize}(\omega') \in A\}$ for a set of states A.

Lemma 2. *Given a set $\Omega \in \mathbb{P}(\Sigma^* \times \Sigma)$ of lists of past states paired with current states, a CoreIVL statement C, and a function $\mathcal{S}$ mapping elements from Ω to sets of states, if for all $(l, \omega) \in \Omega$ we have*

(1) stable(ω), and
(2) $\langle C, \omega \rangle \rightarrow_\Delta \mathcal{S}(l, \omega)$,

then $\Delta \vdash [\ll\{\omega \mid (l, \omega) \in \Omega\}\gg]\ C\ [\ll\bigcup_{(l,\omega)\in\Omega} \mathcal{S}(l, \omega)\gg]$.

An element $([\omega_0, \dots, \omega_n], \omega_{n+1}) \in \Omega$ represents all the intermediate states of one execution up to now, which we use to resolve the future angelism. The function $\mathcal{S}$ maps each such element to a set of states that can be reached from ω_{n+1} by executing C. Intuitively, the precondition collects all the current states from Ω, and the postcondition collects all the states they can reach by executing C. The proof proceeds by structural induction over the statement C.

The reason for tracking sequences of past states. The reader might be wondering why Lemma 2 keeps track of the list of all past states, instead of only keeping track of the current state. The reason is that only keeping track of the current state would not allow us to prove the inductive case for sequential composition. To understand why, *assume* that Ω is *instead* a set of *single states*, and $\mathcal{S}$ is a function from *single states* to a set of states. Consider proving the inductive case for the sequential composition, *i.e.*, for $C \triangleq (C_1; C_2)$. Assume that, in this scenario, we are given $\Omega = \{\omega_A, \omega_B\}$, and that $\mathcal{S}$ is such that $\mathcal{S}(\omega_A) = \{\omega'_A\}$ and $\mathcal{S}(\omega_B) = \{\omega'_B\}$. From assumption (2) in Lemma 2, we know that $\langle C_1; C_2, \omega_A \rangle \rightarrow_\Delta \{\omega'_A\}$ and $\langle C_1; C_2, \omega_B \rangle \rightarrow_\Delta \{\omega'_B\}$ hold. It might be the case that executing C_1 in either ω_A or ω_B yields the same set of states $\{\omega'\}$, *i.e.*, $\langle C_1, \omega_A \rangle \rightarrow_\Delta \{\omega'\}$ and $\langle C_1, \omega_B \rangle \rightarrow_\Delta \{\omega'\}$, but that the angelic non-determinism when executing C_2 in state ω' was resolved differently in both executions, leading to ω'_A in the execution from ω_A and ω'_B in the execution from ω_B. More concisely, the executions of $C_1; C_2$ in ω_A and ω_B might have

been constructed as follows:

$$\langle C_1, \omega_A \rangle \rightarrow_\Delta \{\omega'\} \wedge \langle C_2, \omega' \rangle \rightarrow_\Delta \{\omega'_A\} \Rightarrow \langle C_1; C_2, \omega_A \rangle \rightarrow_\Delta \{\omega'_A\}$$

$$\langle C_1, \omega_B \rangle \rightarrow_\Delta \{\omega'\} \wedge \langle C_2, \omega' \rangle \rightarrow_\Delta \{\omega'_B\} \Rightarrow \langle C_1; C_2, \omega_B \rangle \rightarrow_\Delta \{\omega'_B\}$$

In this case, our intermediate set of states between C_1 and C_2 is $\Omega \triangleq \{\omega'\}$. To apply our induction hypothesis for C_2, we need to find a function $\mathcal{S}_2$ that maps ω' to both $\{\omega'_A\}$ and $\{\omega'_B\}$, as required by the assumption (2) in the lemma, which is not possible.

To solve this issue, we explicitly keep track of all past states. In this way, our intermediate set of states for the previous example is $\Omega \triangleq \{([\omega_A], \omega'), ([\omega_B], \omega')\}$, which allows us to define a function $\mathcal{S}_2$ such that $\mathcal{S}_2([\omega_A], \omega') = \{\omega'_A\}$ and $\mathcal{S}_2([\omega_B], \omega') = \{\omega'_B\}$, allowing us to apply our induction hypothesis and prove the sequential composition case.

Completeness. To show that our operational and axiomatic semantics are equivalent, we also prove the following completeness property (whose proof is less involved than for soundness):

Theorem 5 (Completeness). *Assume* $\Delta \vdash [P]\, C\, [Q]$, *and let* $\omega \in P$ *such that* $stable(\omega)$. *Then there exists* S *such that* $\langle C, \omega \rangle \rightarrow_\Delta S$ *and* $S \subseteq Q$.

3.4 ViperCore: Instantiating CoreIVL with Viper

To show the practical usefulness of CoreIVL, we instantiated it for the Viper language. We call this instantiation ViperCore, and we use it in §4 and §5. To instantiate the framework presented in this section, one needs (1) an IDF algebra, (2) a type of custom statements C', (3) operational and axiomatic semantic rules for each custom statement, and (4) proofs that those operational and axiomatic semantic rules are compatible with our framework (*i.e.*, soundness and completeness for the custom semantic rules, and a proof that the operational semantics of custom statements preserves stability).

We instantiate (1) with the IDF algebra Σ_{IDF} defined in §3.1, where the set L of heap locations is the set of pairs of a reference and a field (represented by a string). For (2), we add field assignments as $C' ::= (e_1.f := e_2)$, where e_1 and e_2 are semantic expressions that evaluate to a reference and a value, respectively, and f is a field. The field assignment $e_1.f := e_2$ is deterministic. In an initial state $(\sigma, (h, \pi))$, it reduces to the singleton set $\{(\sigma, (h[(r, f) \mapsto v], \pi))\}$ if e_1 evaluates to a reference r, e_2 evaluates to a value v, and $\pi((r, f)) = 1$. This semantics is reflected both in its corresponding operational and axiomatic semantic rules (3), and the associated proofs (4) are straightforward.

Moreover, we have also connected the deep embedding of the Viper language developed by Parthasarathy et al. [2024] (which we leverage in the next section) to ViperCore, by defining a function $\downarrow C$ that converts their *syntactic* statements, expressions and assertions into *semantic* ViperCore statements, expressions and assertions.

4 Back-End Soundness

In this section, we show how our framework enables formalizing the soundness of different back-end verifiers. We prove the soundness of two fundamentally different verification algorithms commonly used in practice: symbolic execution and verification condition generation. We connect both to the *same* instantiation of CoreIVL, namely ViperCore introduced in §3.4. This demonstrates that CoreIVL's semantics can accommodate fundamentally different verification algorithms.

Symbolic execution is a common kind of verification algorithm used in separation logic-based verifiers [Berdine et al. 2005b; Jacobs et al. 2011; Santos et al. 2020]. §4.1 introduces a symbolic execution back-end for ViperCore. Its design follows Viper's symbolic execution back-end [Schwerhoff 2016], but it is formalized as a function inside Isabelle/HOL. The main result of §4.1 is a soundness

$$\sigma : \mathsf{SymState} ::= \{\mathsf{store} : \mathsf{Var} \rightharpoonup \mathsf{SymExpr}, \mathsf{pc} : \mathsf{SymExpr}, \mathsf{heap} : \mathsf{List}(\mathsf{Chunk})\}$$

$$t ::= x \mid l \mid \odot t \mid t \oplus t \text{ where } \odot \in \{\neg, -, \ldots\} \text{ and } \oplus \in \{\wedge, \vee, =, +, -, \ldots\}$$

$$c : \mathsf{Chunk} ::= \{\mathsf{recv} : \mathsf{SymExpr}, \mathsf{field} : \mathsf{FieldName}, \mathsf{perm} : \mathsf{SymExpr}, \mathsf{val} : \mathsf{SymExpr}\}$$

$$\mathsf{sexec}\ \sigma\ C\ K \triangleq \begin{cases} \mathsf{sproduce}\ \sigma\ A\ K & \text{if } C = \mathbf{inhale}\ A \\ \mathsf{sconsume}\ \sigma\ A\ (\lambda\sigma.\ \mathsf{scleanup}\ \sigma\ K) & \text{if } C = \mathbf{exhale}\ A \\ \mathsf{sexp}\ \sigma\ e\ (\lambda\sigma\ t.\ \mathsf{sexec}\ \mathsf{pc\_add}(\sigma, t)\ C_1\ K & \text{if } C = (\text{if } e \text{ then } C_1 \text{ else } C_2) \\ \quad \wedge\ \mathsf{sexec}\ \mathsf{pc\_add}(\sigma, \neg t)\ C_2\ K) & \\ \ldots & \end{cases}$$

$$\mathsf{sproduce}\ \sigma\ (\mathbf{acc}(e_r.f, e_p))\ K \triangleq \mathsf{sexp}\ \sigma\ e_r\ (\lambda\sigma\ t_r.\ \mathsf{sexp}\ \sigma\ e_p\ (\lambda\sigma\ t_p.\ \mathsf{chunk\_add}\ \sigma\ \{t_r, f, t_p, \mathsf{fresh}\}\ K))$$

$$\mathsf{sconsume}\ \sigma\ (\mathbf{acc}(e_r.f, \_))\ K \triangleq \mathsf{sexp}\ \sigma\ e_r\ (\lambda\sigma\ t_r.\ \mathsf{extract}\ \sigma\ t_r\ f\ \_\ (\lambda\sigma\ c.\ \mathsf{chunk\_add}\ \sigma\ c\{\mathsf{perm} := c.\mathsf{perm}/2\}\ K))$$

Fig. 9. Symbolic states and excerpts of sexec, sproduce, and sconsume. The full definition is in the extended version [Dardinier et al. 2024a, Section A].

proof of this symbolic execution against the operational semantics of ViperCore, showing how CoreIVL is general enough to justify widely-used symbolic execution algorithms.

In §4.2 we connect ViperCore to the formalization by Parthasarathy et al. [2024] of Viper's verification condition generation (VCG) back-end, which translates an input Viper program to Boogie.[8] This formalization includes a formal operational semantics of Viper that we call *VCGSem*. Unlike ViperCore, which is designed to capture the verification algorithms of multiple back-ends, VCGSem is specific to the verification algorithm of the VCG back-end. For example, VCGSem uses a total heap (*i.e.*, all possible locations on the heap store a value), while ViperCore is based on a partial heap (which is important to capture existing symbolic execution algorithms). Moreover, VCGSem uses (constrained) *demonic* choice when exhaling wildcard permissions, while ViperCore uses angelic choice. Despite these differences, we show that ViperCore's (and thus also CoreIVL's) operational semantics is general enough to capture VCGSem, which embodies Viper's VCG back-end.

We have chosen these two back-ends since they implement very different proof search algorithms: The symbolic execution algorithm manipulates a *symbolic state* including a list of heap chunks, while the VCG back-end maps to Boogie code whose operations are embodied by VCGSem, a big-step operational semantics with a total heap. These back-ends show CoreIVL's generality for justifying multiple common verification algorithms. A key aspect that enables this generality is CoreIVL's use of angelic choice. Concretely, the two back-ends use different algorithms for exhaling wildcard permissions (the symbolic execution halves the permission of one heap chunk while VCGSem *demonically* chooses a suitably-constrained permission amount). Yet, CoreIVL can capture both algorithms thanks to its use of angelic choice.

4.1 Symbolic Execution

We formalized a symbolic execution back-end for ViperCore in Isabelle/HOL based on the description of Viper's back-end by Schwerhoff [2016] while also taking inspiration from the (on paper) formalization of symbolic execution by Zimmerman et al. [2024].

---

[8]This work provides a proof-producing version of Viper's VCG back-end that generates a certificate in Isabelle for each successful verification, but not a general soundness proof.

Symbolic states. The symbolic state tracked during verification is defined in Fig. 9. It consists of the following components:[9] (1) A *symbolic store* (store) mapping variables to symbolic expressions, (2) a *path condition* (pc)—which is a symbolic expression tracking logical facts that hold in the current branch of the program—, and (3) a *symbolic heap* (heap) given by a list of heap chunks. *Symbolic expressions t* consist of symbolic variables x, literals l (*e.g.*, for concrete integers, booleans or permission amounts), unary operations $\odot\, t$, and binary operations $t \oplus t$. We define a function $\mathrm{pc\_add}(\sigma, t)$ that adds the (boolean) symbolic expression t to the path condition of σ.

The most crucial part of symbolic states is the symbolic heap. As is common [Berdine et al. 2005a; Jacobs et al. 2011; Schwerhoff 2016], we represent the symbolic heap as a list of (heap) chunks. Conceptually, each heap chunk corresponds to an $\mathbf{acc}(e_r.f, e_p)$ resource, which we call an acc-resource in this section, together with an associated value. Concretely, a chunk c is a record with four fields, as shown in Fig. 9. recv and field describe the heap location that the chunk belongs to, perm describes the permission of the chunk, and val gives the (symbolic) value of the heap location. A symbolic heap is a list of chunks. Note that this list can contain multiple chunks for the same location (*cf.* state consolidation, described shortly).

Defining the symbolic execution. Our symbolic execution is defined via the sexec function for symbolically executing a statement C. It delegates calls to: the sproduce function (for inhaling an assertion A), the sconsume function (for exhaling an assertion A), the scleanup function (for removing empty heap chunks after exhaling an assertion), and the sexp function (for symbolically evaluating an expression e). Each of these functions are formalized as functions in Isabelle/HOL and can be executed inside the prover to verify a concrete program. The parts of these functions relevant to this chapter are shown in Fig. 9. The full definition can be found in the extended version [Dardinier et al. 2024a, Section A]. Following Schwerhoff [2016], these functions are written in continuation passing style with continuation K. This allows us to easily split the verification in multiple branches as shown *e.g.*, by the if-case of sexec. We now highlight the most important aspects of the symbolic execution.

Representing different state consolidation algorithms. After inhaling an acc-resource and adding it to the list of heap chunks, the symbolic execution might try to merge chunks for the same location and deduce additional information (*e.g.*, that for chunks of the same location their permissions sum to at most 1 and their values match). This process, called *state consolidation* [Schwerhoff 2016], is incorporated into the chunk_add function, used to model inhaling an acc-resource during sproduce:

$$\mathrm{chunk\_add}\ \sigma\ c\ K \triangleq \mathrm{consolidate}\ \sigma\{\mathrm{heap} := c :: \sigma.\mathrm{heap}\}\ K$$

Since there are many possible ways to implement state consolidation [Schwerhoff 2016] (*e.g.*, merging chunks eagerly or lazily), we do not prescribe a specific implementation of the consolidate function, but instead characterize consolidate *semantically*:[10]

$$\mathrm{consolidate}\ \sigma\ K \triangleq \forall \omega.\ \omega \sim_{\mathsf{sym}} \sigma \Rightarrow \exists \sigma'.\ \omega \sim_{\mathsf{sym}} \sigma' \wedge K\ \sigma'$$

Concretely, when executing consolidate, one is given a ViperCore state ω related to the current symbolic execution state σ (using the $\sim_{\mathsf{sym}}$ relation) and one can pick an arbitrary new state σ' as long as it is related to the same ViperCore state ω. Intuitively, $\omega \sim_{\mathsf{sym}} \sigma$ is defined by stating that there exists a mapping from symbolic variables to concrete values, which can be simply extended to a mapping from σ to ω. The existential quantifier allows us to represent many different state consolidation algorithms. However, this generality also means that consolidate cannot be executed

---

[9]For simplicity, we omit components for generating fresh symbolic variables and tracking type information.

[10]The actual definition of consolidate is slightly different to decouple the definition of the symbolic execution and ViperCore.

directly. Instead, one can provide a concrete algorithm and prove it sound against consolidate (our implementation uses the trivial algorithm that does not consolidate at all). However, the soundness proof of our symbolic execution works for any valid consolidation algorithm.

Soundness. We prove sexec sound against the operational semantics of ViperCore:[11]

Theorem 6 (Soundness of sexec). *For each (syntactic) statement C, ViperCore state ω and symbolic state σ related via $\omega \sim_{sym} \sigma$, if sexec σ C K evaluates to true, then $\downarrow C$ is correct for the initial state ω.*

$\downarrow C$ is the compilation function from syntactic statements to ViperCore statements described in §3.4. The operational semantics of CoreIVL is well-suited for this soundness proof since the symbolic execution also traverses the statements in an operational way, and it is straightforward to relate one ViperCore state to one symbolic execution state via $\omega \sim_{sym} \sigma$.

Soundness of exhaling wildcards via angelic choice. Let us highlight the most interesting part of this soundness proof: exhaling wildcards. Exhaling assertions is handled by the sconsume function in Fig. 9. When exhaling an acc-resource with a wildcard permission amount, sconsume finds and removes a matching chunk from the symbolic heap using the extract function.[12] Then it adds the chunk back with its permission amount halved. Representing this algorithm directly in ViperCore would be impossible since there might be multiple heap chunks for the same location and thus the amount of permissions removed depends on the structure of the symbolic heap. This structure is not visible in ViperCore, which tracks only a single concrete heap. However, we can still prove this algorithm sound against ViperCore. The angelic choice in the operational semantics allows us to pick *any* non-zero permission amount to remove when constructing the ViperCore execution, in particular, the amount that was chosen by the execution of sexec. This shows how angelic choice gives CoreIVL the flexibility to be used in the soundness proof for different verification algorithms, even some that cannot be represented directly in the CoreIVL.

4.2 Verification Condition Generation

We now describe how we connect the ViperCore instantiation of CoreIVL to the VCGSem formalization of Viper's VCG introduced by Parthasarathy et al. [2024]. VCGSem is expressed as an operational big-step semantics $\langle C, \sigma_t \rangle \rightarrow_{VCG} r$. Here, C is a (deeply embedded) Viper statement, σ_t the initial VCGSem state consisting of a total heap (mapping all locations to values) and a permission mask (mapping all locations to permission amounts), and r is an *outcome*, which can be either *failure* F, *magic* M, or a *normal outcome* $N(\sigma_t')$.[13] The key result of Parthasarathy et al. [2024] is that for each successful verification run of the VCG algorithm, they provide a proof that the VCGSem execution does not fail: $\neg(\langle C, \sigma_t \rangle \rightarrow_{VCG} F)$.

What makes the connection between VCGSem and ViperCore interesting is that VCGSem makes various design choices that are specific to the Viper back-end that it was designed to represent. For instance, VCGSem defines the **exhale** of a wildcard to demonically remove a non-zero permission amount smaller than the currently held amount, which precisely mimics Viper's VCG. Morever, VCGSem chooses a total heap representation for the Viper states, where *all* locations store a value (VCGSem checks that only locations with non-zero permission are accessed), because this is how Viper's VCG back-end represents the heap. In contrast, ViperCore uses a more standard partial heap introduced in §3.1. By proving VCGSem sound against ViperCore, we show that CoreIVL as a general semantics for verification algorithms can capture this preexisting verification algorithm.

---

[11]We omit side-conditions about typing to avoid clutter.

[12]Similar to consolidate, extract is characterized semantically and we provide a default implementation of extract that queries Isabelle/HOL's solvers to find the first matching chunk.

[13]We omit typing contexts in this section to avoid clutter.

The most significant challenge in the proof connecting VCGSem and ViperCore is the difference in their heap representations. We explain this challenge and our solutions next.

Total vs. partial heap. The seemingly superficial difference between VCGSem's total heap and ViperCore's partial heap has far-reaching ramifications: fundamentally it means that a ViperCore execution does not correspond to a single VCGSem execution but a *set* of VCGSem executions.

The reason for this mismatch is in the semantics of **exhale**. When exhaling all permissions to a location and later inhaling permissions to this location again, a Viper semantics needs to pick a *fresh* value for the location such that one cannot unsoundly assume that the value remained unchanged between the **inhale** and the **exhale**. This requirement is naturally expressed with the partial heap of ViperCore: when exhaling all permissions to a location in ViperCore, the location is removed from the partial heap and when new permissions for the location are inhaled, it gets re-added with a (non-deterministically chosen) fresh value. However, since VCGSem uses a total heap, it cannot *remove* locations. Instead, VCGSem non-deterministically assigns these locations new values *after the* **exhale** and leaves the heap unchanged in the **inhale**. Consequently, VCGSem and ViperCore apply (demonic) non-deterministic choice at different program points: VCGSem already picks a fresh value during the **exhale**, while ViperCore chooses it during the **inhale**. To address this mismatch[14], we relate a ViperCore execution not to a single VCGSem execution but to a set of VCGSem executions that represent all possible choices for the demonic non-determinism.

Soundness. We prove the following soundness statement for VCGSem:[15]

Theorem 7 (Soundness of VCGSem). *For all (syntactic) statements C and ViperCore states ω, if we have $\neg(\langle C, \sigma_t \rangle \rightarrow_{\mathsf{VCG}} F)$ for all VCGSem states σ_t such that $\omega \sim_{\mathsf{VCG}} \sigma_t$, then $\downarrow C$ is correct for the state ω.*

Intuitively, this theorem allows us to transform a proof about a successful verification by the VCG back-end into a verification proof according to the ViperCore semantics. Note that the theorem relates a single ViperCore execution to a *set* of VCGSem executions since the relation $\omega \sim_{\mathsf{VCG}} \sigma_t$ relates a ViperCore state ω to multiple VCGSem states σ_t representing the different choices for the demonic non-determinism. (Otherwise, $\omega \sim_{\mathsf{VCG}} \sigma_t$ is similar to $\omega \sim_{\mathsf{sym}} \sigma$ from §4.1, but adapted for the different notion of states used by VCGSem.) In fact, to prove Thm. 7 via induction, we need to prove a stronger lemma that also requires us to construct all possible VCGSem executions for the statement corresponding to the ViperCore execution.

Summary. We have demonstrated in this section how CoreIVL's operational semantics helps us solve Challenge 2, by being general enough to capture the two predominant verification algorithms back-ends implemented in practice: our new formalization of symbolic execution in §4.1 and the preexisting formalization of Viper's VCG back-end [Parthasarathy et al. 2024] in §4.2.

5 Front-End Soundness

In this section, we show how our axiomatic semantics addresses Challenge 3 from §1, by formalizing and proving sound a concrete front-end translation into ViperCore for a parallel programming language ParImp with loops, shared memory, and dynamic memory allocation and deallocation. We define the language and an IDF-based program logic in §5.1. In §5.2, we define the translation of annotated ParImp programs into ViperCore and prove it sound using the axiomatic semantics

---

[14]The mismatch could also be addressed by changing VCGSem to assign a fresh value during **inhale**. However, our goal is to capture the verification algorithms of *existing* back-ends.

[15]For readability, we omit some technical assumptions about stability of ω and well-typedness.

FRAME
$$\frac{\Delta \vdash_{CSL} [P]\ C\ [Q] \quad selfFraming(P) \quad selfFraming(F) \quad fv(F) \cap mod(C) = \emptyset}{\Delta \vdash_{CSL} [P * F]\ C\ [Q * F]}$$

PAR
$$\frac{mod(C_l) \cap (fv(C_r) \cup fv(Q_r)) = \varnothing \quad mod(C_r) \cap (fv(C_l) \cup fv(Q_l)) = \varnothing}{\Delta \vdash_{CSL} [P_l]\ C_l\ [Q_l] \quad \Delta \vdash_{CSL} [P_r]\ C_r\ [Q_r] \quad selfFraming(P_l) \quad selfFraming(P_r)} {\Delta \vdash_{CSL} [P_l * P_r]\ C_l\ ||\ C_r\ [Q_l * Q_r]}$$

SEQ
$$\frac{\Delta \vdash_{CSL} [P]\ C_1\ [R] \quad \Delta \vdash_{CSL} [R]\ C_2\ [Q]}{\Delta \vdash_{CSL} [P]\ C_1;C_2\ [Q]}$$

CONS
$$\frac{\Delta \vdash_{CSL} [P']\ C\ [Q'] \quad P \models P' \quad Q' \models Q}{\Delta \vdash_{CSL} [P]\ C\ [Q]}$$

IF
$$\frac{\Delta \vdash_{CSL} [P \wedge b]\ C_1\ [Q] \quad \Delta \vdash_{CSL} [P \wedge \neg b]\ C_2\ [Q]}{\Delta \vdash_{CSL} [P]\ \mathbf{if}(b)\ \{C_1\}\ \mathbf{else}\ \{C_2\}\ [Q]}$$

ALLOC
$$\frac{r \notin fv(e)}{\Delta \vdash_{CSL} [\top]\ \mathbf{r} := \mathbf{alloc}(e)\ [\mathbf{acc}(\mathbf{r.v}) * \mathbf{r.v} = e]}$$

WHILE
$$\frac{\Delta \vdash_{CSL} [I \wedge b]\ C\ [I]}{\Delta \vdash_{CSL} [I]\ \mathbf{while}\ (b)\ \{C\}\ [I \wedge \neg b]}$$

LOAD
$$\frac{P \models \mathbf{acc}(\mathbf{r.v}, \_)}{\Delta \vdash_{CSL} [P]\ \mathbf{x} := \mathbf{r.v}\ [\exists u.\, P[u/x] * \mathbf{x} = \mathbf{r.v}]}$$

STORE
$$\Delta \vdash_{CSL} [\mathbf{acc}(\mathbf{r.v})]\ \mathbf{r.v} := e\ [\mathbf{acc}(\mathbf{r.v}) * \mathbf{r.v} = e]$$

FREE
$$\Delta \vdash_{CSL} [\mathbf{acc}(\mathbf{q.v})]\ \mathbf{free}(\mathbf{q})\ [\top]$$

ASSIGN
$$\Delta \vdash_{CSL} [P[x/e]]\ \mathbf{x} := e\ [P]$$

SKIP
$$\Delta \vdash_{CSL} [P]\ \mathbf{skip}\ [P]$$

Fig. 10. Inference rules of our IDF-based CSL.

of ViperCore. While the soundness proof is specific to this translation, it highlights key reusable ingredients and demonstrates how our axiomatic semantics for CoreIVL makes such proofs simple.

5.1 An IDF-Based Concurrent Separation Logic

Our parallel programming language ParImp is defined as

$$C ::= x := e \mid x := r.v \mid r.v := e \mid r := \mathbf{alloc}(e) \mid \mathbf{free}(r) \mid C;C \mid \mathbf{if}(b)\ \{C\}\ \mathbf{else}\ \{C\} \mid C||C \mid \mathbf{while}\ (b)\ \{C\} \mid \mathbf{skip}$$

C ranges over ParImp statements, e over arithmetic expressions, b over boolean expressions, x over integer variables, r over reference variables, and v is a fixed field. We consider objects with a unique field v for simplicity; extending our work to support multiple fields is straightforward. The statement $x := r.v$ loads the value of the field v of the reference r into the variable x, while $r.v := e$ stores the value of the expression e in the field v of the reference r. The statement $r := \mathbf{alloc}(e)$ allocates a new reference with the value of the expression e for the field v, and $\mathbf{free}(r)$ deallocates the reference r. The other statements are standard. We use a standard small-step semantics $\langle C, \sigma \rangle \rightarrow \langle C', \sigma' \rangle$ where σ and σ' are pairs of a store (a partial mapping from variables to values) and a heap (a partial mapping from pairs of an address and a field to values).

An IDF-based program logic for ParImp. We build and prove sound a program logic analogous to CSL for ParImp based on our IDF state model Σ_{IDF} (defined in §3.1). Our framework also supports standard separation logic, but connecting an IDF logic to ViperCore allows us to focus on the most interesting aspects of the soundness proof.

Our program logic judgment is written $\Delta \vdash_{CSL} [P]\ C\ [Q]$, where P and Q are ViperCore assertions (*i.e.*, sets of IDF states). The most important rules of our program logic are given in Fig. 10. The rules SEQ, CONS, IF, WHILE, FREE, ASSIGN, and SKIP, are standard. The rules ALLOC, STORE, LOAD, FRAME, and

$[\![r := \mathbf{alloc}(e)]\!]$	$\triangleq ((\mathbf{havoc}\ r; \mathbf{inhale}\ \mathbf{acc}(r.v) * r.v = e), \varnothing)$	$[\![\mathbf{skip}]\!]$	$\triangleq (\mathbf{skip}, \varnothing)$
$[\![\mathbf{free}(r)]\!]$	$\triangleq (\mathbf{exhale}\ \mathbf{acc}(r.v), \varnothing)$	$[\![x := e]\!]$	$\triangleq (x := e, \varnothing)$
$[\![C_1; C_2]\!]$	$\triangleq (([\![C_1]\!].1; [\![C_2]\!].1), ([\![C_1]\!].2 \cup [\![C_2]\!].2))$	$[\![r.v := e]\!]$	$\triangleq (r.v := e, \varnothing)$
$[\![\mathbf{if}(b)\ \{C_1\}\ \mathbf{else}\ \{C_2\}]\!]$	$\triangleq (\mathbf{(if}(b)\ \{[\![C_1]\!].1\}\ \mathbf{else}\ \{[\![C_2]\!].1\}), ([\![C_1]\!].2 \cup [\![C_2]\!].2))$	$[\![x := r.v]\!]$	$\triangleq (x := r.v, \varnothing)$

$$[\![C_l\ ||\ C_r]\!] \quad \triangleq ((\mathbf{exhale}\ P_l * P_r; \mathbf{havoc}\ mod(C_l) \cup mod(C_r); \mathbf{inhale}\ Q_l * Q_r),$$
$$\{\mathbf{inhale}\ P_l; [\![C_l]\!].1; \mathbf{exhale}\ Q_l\} \cup \{\mathbf{inhale}\ P_r; [\![C_r]\!].1; \mathbf{exhale}\ Q_r\} \cup [\![C_l]\!].2 \cup [\![C_r]\!].2)$$

$$[\![\mathbf{while}\ (b)\ \{C\}]\!] \quad \triangleq ((\mathbf{exhale}\ I; \mathbf{havoc}\ mod(C); \mathbf{inhale}\ I \wedge \neg b),$$
$$\{\mathbf{inhale}\ I \wedge b; [\![C]\!].1; \mathbf{exhale}\ I\} \cup [\![C]\!].2)$$

Fig. 11. Front-end translation from ParImp to ViperCore. The translation function $[\![\_]\!]$ takes as input an annotated ParImp statement C and returns a pair of a ViperCore statement and a set of ViperCore statements. We write $[\![C]\!].1$ and $[\![C]\!].2$ to denote its first and second components, respectively. Assertions P_l, P_r, Q_l, and Q_r for the parallel composition and I for the while loop are annotations provided by the user, which are all required to be self-framing. The notation $\mathbf{havoc}\ V$, where V is a set of variables $\{x_1, \ldots, x_n\}$, is a shorthand for $\mathbf{havoc}\ x_1; \ldots; \mathbf{havoc}\ x_n$.

Par are analogous to the standard CSL rules, but adapted to our IDF setting. In particular, the rule Frame requires the precondition P and the frame F to be self-framing. Without this restriction, one could for example use $P \triangleq (\mathbf{acc}(r.v))$ and $F \triangleq (r.v = 5)$ to unsoundly derive the invalid triple $\Delta \vdash_{CSL} [(\mathbf{acc}(r.v)) * (r.v = 5)]\ r.v := 3\ [(\mathbf{acc}(r.v) * r.v = 3) * (r.v = 5)]$ (whose postcondition is not satisfiable) using the rules Frame and Store. Similarly, the rule Par requires the preconditions P_l and P_r to be self-framing. Finally, the rule Load allows arbitrary preconditions P, as long as P asserts some permission to read $r.v$.

We have proved in Isabelle the soundness of this IDF-based program logic, which we state as follows (the proof of this theorem is an adaption of the proof from Vafeiadis [2011] to our IDF setting):

Theorem 8 (Adequacy). *Let C be a well-typed program, and P and Q be predicates on ParImp states (i.e., without permissions). If the triple $\Delta \vdash_{CSL} [P]\ C\ [Q]$ holds, and if σ is a well-typed state such that $P(\sigma)$, then executing C in the state σ will not abort nor encounter any data race, and for all σ' such that $\langle C, \sigma \rangle \rightarrow^* \langle \mathbf{skip}, \sigma' \rangle$, we have $Q(\sigma')$.*

5.2 A Sound Front-End Translation

Building on the previously-defined IDF-based program logic, we define a standard front-end translation from ParImp programs with annotations into ViperCore programs, shown in Fig. 11. This translation was illustrated in the example in Fig. 3 from §2. The translation function $[\![\_]\!]$ takes as input an annotated ParImp statement C and yields a *pair of a ViperCore statement and a set of ViperCore statements*. The first component, written $[\![C]\!].1$, corresponds to the main translation of C, while the second component, written $[\![C]\!].2$, corresponds to the set of auxiliary Viper methods generated by the translation along the way. Auxiliary methods are generated for loops and parallel compositions only. Methods l and r in Fig. 3 are examples of such auxiliary methods.

The translation of field and variable assignments is straightforward. The translation of sequential composition and conditional statements is also straightforward since they use the corresponding sequential composition and conditional statements of ViperCore, and collect the auxiliary methods generated by the translation of the sub-statements. The translation of allocation and deallocation statements corresponds to the rules Alloc and Free from Fig. 10.

The translation of parallel composition and while loops is more involved, but they follow the same pattern. First, the premises of the relevant rules (Par and While) are checked by generating auxiliary methods, which first inhale the relevant precondition, then translate the relevant statement, and finally exhale the relevant postcondition. For example, the premise $\Delta \vdash_{CSL} [I \wedge b]\ C\ [I]$ of the rule While is checked by generating the auxiliary method **inhale** $I \wedge b$; $[\![C]\!].1$; **exhale** I. We call this pattern the *inhale-translation-exhale* pattern. Then, the main translation follows the conclusion of the rule, by exhaling the precondition, havocking the modified variables, and inhaling the postcondition. For example, the main translation of the loop **while** (b) $\{C\}$ is (**exhale** I; **havoc** $mod(C)$; **inhale** $I \wedge \neg b$), reflecting the conclusion $\Delta \vdash_{CSL} [I]$ **while** (b) $\{C\}$ $[I \wedge \neg b]$ of the rule *While*. We call this pattern, which we have already seen in §2.4, the *exhale-havoc-inhale* pattern. Those two patterns are not specific to our translation, but are general patterns that can be found in many front-end translations.

Soundness. We assume that the ParImp statement C we want to verify is annotated with a precondition P and a postcondition Q. In this case, we add **inhale** P before the main translation (as we did in Fig. 3), and **exhale** Q afterwards, following the inhale-translation-exhale pattern. Our complete front-end translation yields the set of ViperCore statements $\{$**inhale** P; $[\![C]\!].1$; **exhale** $Q\} \cup [\![C]\!].2$. Our translation is sound, as stated in the following theorem. We say that a ViperCore statement C_v is valid *w.r.t. the axiomatic semantics*, which we write $valid_{Ax}(C_v)$, iff $\exists B.\ \Delta \vdash [\top]\ C_V\ [B]$.

Theorem 9 (Soundness of the front-end translation). *Let C be a front-end statement, and P and Q be assertions. If (1) the axiomatic semantics triple $\Delta \vdash [P]\ [\![C]\!].1\ [Q]$ holds, and (2) all ViperCore statements in $[\![C]\!].2$ are valid w.r.t. the axiomatic semantics, then $\Delta \vdash_{CSL} [P]\ C\ [Q]$ holds.*

To prove this theorem, we show that the translation of every front-end statement C is *backward-convertible* (or *convertible* in short), which we write as $convertible(C)$. Intuitively, this means that if the translation of the front-end statement into ViperCore is valid (including all auxiliary ViperCore methods) then we can convert the axiomatic semantics triple $\Delta \vdash_{CSL} [P]\ [\![C]\!].1\ [Q]$ into a front-end triple $\Delta \vdash_{CSL} [P]\ C\ [Q]$. We formally express this property as follows:

$$convertible(C) \triangleq \left(\forall P, Q.\ ((\forall C_v \in [\![C]\!].2.\ valid_{Ax}(C_v)) \wedge \Delta \vdash [P]\ [\![C]\!].1\ [Q]) \Rightarrow \Delta \vdash_{CSL} [P]\ C\ [Q]\right)$$

This convertibility property combined with the following lemma allows us to prove Thm. 9:

Lemma 3 (Inhale-translation-exhale pattern). *If (1) convertible(C) holds, (2) all auxiliary methods from $[\![C]\!].2$ are valid w.r.t. the axiomatic semantics, and (3) $\Delta \vdash [P]$ **inhale** A; $[\![C]\!].1$; **exhale** B $[Q]$ holds, then $\Delta \vdash_{CSL} [P * A]\ C\ [B * Q]$ holds.*

PROOF. By inverting the rules SeqAx, InhaleAx, and ExhaleAx, we get the existence of R such that (a) $\Delta \vdash [P * A]\ [\![C]\!].1\ [R]$ holds and (b) $R \models B * Q$. By applying *convertible*(C), and from (2) and (a), we get $\Delta \vdash_{CSL} [P * A]\ C\ [R]$. We conclude by combining (b) with the rule Cons.   □

The proof of this lemma is straightforward thanks to CoreIVL's *axiomatic* semantics. Relating CSL to an operational IVL semantics would require substantially more effort to re-prove standard reasoning principles, which we prove once and for all in the equivalence proof of the two IVL semantics.

We now need to prove *convertible*(C) for all C, which we do by structural induction. The inductive cases for most statements are straightforward; the interesting cases are allocation, deallocation, parallel compositions, and while loops. As explained above, the main translation of those statements follows the same exhale-havoc-inhale pattern, which we have already seen in §2.4, and prove below:

Lemma 4 (Exhale-havoc-inhale). *Let P and Q be self-framing assertions.*[16] *Assume that* $\Delta \vdash [A]$ ***exhale*** P; ***havoc*** x_1; . . . ; ***havoc*** x_n; ***inhale*** Q $[B]$ *holds, where* $\{x_1, \ldots, x_n\} = mod(C)$. *If* $\Delta \vdash_{\mathcal{L}} [P] \ C \ [Q]$ *holds, and if $\mathcal{L}$ has a frame rule and a consequence rule, then* $\Delta \vdash_{\mathcal{L}} [A] \ C \ [B]$ *holds.*

PROOF. By inverting the rule SEQAX, we obtain F such that (a) $\Delta \vdash [F]$ ***inhale*** Q $[B]$ and (b) $\Delta \vdash [A]$ ***exhale*** P; ***havoc*** x_1; . . . ; ***havoc*** x_n $[F]$ hold. From (b), by inverting the rules SEQAX and HAVOCAX, we obtain an assertion R such that (c) $\Delta \vdash [A]$ ***exhale*** P $[R]$ holds, (d) $fv(F) \cap \{x_1, \ldots, x_n\} = \emptyset$, and (e) $R \models F$.[17] By applying the frame rule with F and $\Delta \vdash_{\mathcal{L}} [P] \ C \ [Q]$, where the side condition is justified by (d), we get $\Delta \vdash_{\mathcal{L}} [P * F] \ C \ [Q * F]$. Finally, we obtain $B = F * Q$ from (a) (by inverting the rule INHALEAX), and $A \models P * F$ from (c) (by inverting the rule EXHALEAX) and (e); applying the consequence rule yields $\Delta \vdash_{\mathcal{L}} [A] \ C \ [B]$. □

This proof shows that, in this pattern, the role of the ***exhale*** statement, followed by a sequence of ***havoc*** statements, is to compute (implicity) the suitable frame for the front-end statement. The ***inhale*** statement afterwards then adds the postcondition of the front-end statement to the frame.

convertible(***free***(r)) and *convertible*($r :=$ ***alloc***(e)) follow directly from the lemma above, by observing that ***inhale*** $\top$ and ***exhale*** $\top$ are equivalent to ***skip*** (and so omitted when encoding).

To prove *convertible*(***while*** (b) $\{C\}$) (assuming C is convertible), we first apply Lemma 3 on the auxiliary method ***inhale*** $I \wedge b$; $[\![C]\!].1$; ***exhale*** I to get $\Delta \vdash_{CSL} [I \wedge b] \ C \ [I]$. We then apply the rule WHILE to get $\Delta \vdash_{CSL} [I]$ ***while*** (b) $\{C\}$ $[I \wedge \neg b]$. Finally, we conclude by applying Lemma 4 on the main translation (***exhale*** I; ***havoc*** $mod(C)$; ***inhale*** $I \wedge \neg b$).

The proof of *convertible*($C_1||C_2$) proceeds similarly, by first applying Lemma 3 on the two auxiliary methods (corresponding to the two premises of the rule PAR), then applying the rule PAR, and concluding by applying Lemma 4. This concludes the proof of *convertible*(C) for all C, and thus the proof of Thm. 9.

Summary. We have demonstrated how the axiomatic semantics from §3.3 helps us solve Challenge 3, by allowing us to prove general lemmas about patterns that are common in front-end translations in a simple and straightforward manner, and to prove the soundness of a concrete front-end translation for a parallel programming language with multiple features not present in the IVL (*e.g.*, loops, dynamic memory allocation and deallocation).

6 Related Work

Semantics of SL-based IVLs. There are two recent formalizations [Parthasarathy et al. 2024; Zimmerman et al. 2024] of subsets of Viper [Müller et al. 2016b]. However, each of them exposes implementation details of a Viper back-end, which does not allow the semantics to be connected to diverse back-ends and also not easily to front-ends. In particular, Parthasarathy et al. [2024] use a total heap representation reflecting the Viper VCG back-end that translates to Boogie (as discussed in §4.2), and Zimmerman et al. [2024] reflect Viper's symbolic execution back-end.

GIL [Maksimovic et al. 2021a], which is the intermediate language of Gillian [Santos et al. 2020; Maksimovic et al. 2021a], is parametric in its (1) state model, which must be provided as a PCM (supporting SL but not IDF states in contrast to CoreIVL), (2) *memory actions* operating on the state model, and (3) *core predicates* describing atomic assertions on the memory such as a SL points-to assertion. For each state instantiation, tool developers targeting GIL must specify *produce* and *consume* actions for each core predicate, which correspond to ***inhale*** and ***exhale*** operations in CoreIVL. Together with instantiated parameters, Maksimovic et al. [2021a] provide an operational semantics for the symbolic execution of GIL. Since the instantiated state effectively reflects the

---

[16]This condition is trivially true for standard SLs.
[17]More precisely, we obtain $F = (\exists x_1, \ldots, x_n. R)$, from which (d) and (e) follow.

symbolic state on which the symbolic execution tool operates, a GIL instantiation essentially represents the back-end semantics. This is in contrast to our CoreIVL, which allows abstracting over multiple back-ends.

Dardinier et al. [2023] define the semantics of a parametric verification language similar to CoreIVL for the purpose of showing formal results on method call inlining in automated SL verifiers. Their semantics is meant to capture IVL *back-ends* with their heuristics. That is, an instantiation reflects a *single* back-end. As a result, in contrast to CoreIVL, their semantics has no angelic nondeterminism. Moreover, their notion of separation algebra to represent states does not support IDF.

Proofs connecting a front-end with an IVL. Summers and Müller [2020] and Wolf et al. [2022] reason about the correctness of translations into a SL-based IVL by providing proof sketches for mapping a correct Viper program to a proof for Hoare triples in the RSL weak memory logic [Vafeiadis and Narayan 2013] and the TaDa logic [da Rocha Pinto et al. 2014], respectively. However, the reasoning is done via proof sketches on paper, which explore only the high-level reasoning principles and thus avoid many of the complexities involved in a fully formal proof. Neither of these works formally reasons about the underlying Viper semantics; they describe the behavior of Viper encodings informally.

Maksimovic et al. [2021b] briefly describe a parametric soundness framework for GIL (the intermediate language of Gillian [Santos et al. 2020; Maksimovic et al. 2021a]). They show that if certain conditions hold on the instantiations of the GIL parameters, then the resulting symbolic execution is sound w.r.t. a concretization function on symbolic states. However, they do not provide an IVL semantics like CoreIVL that abstracts uniformly over multiple back-ends. Additionally, since GIL does not support concurrency [Santos et al. 2020; Maksimovic et al. 2021a], their soundness framework cannot reason about the encoding of front-end languages such as ParImp described in §5. Lööw et al. [2024] present a formal compositional symbolic execution engine inspired by Gillian. In contrast to our work, they focus on supporting both over-approximating and under-approximating reasoning, and do not model an IVL, but only apply their framework to a simple front-end language with a fixed memory model.

There is also work proving the soundness of front-end translations to IVLs not based on SL [Vogels et al. 2009; Backes et al. 2011; Herms 2013; Fortin 2013; Parthasarathy et al. 2024]. However, in contrast to our setting, the corresponding translations do not reflect rules in a front-end program logic. As a result, the soundness proofs work naturally at the level of an operational semantics for the front-end and IVL. Examples include translations from the Dminor data processing language to the Bemol IVL [Backes et al. 2011], from C to the WhyCert IVL (inspired by the Why3 IVL) [Herms 2013], and from Viper to Boogie [Parthasarathy et al. 2024] (in the case of the Viper-to-Boogie translation, Viper is the front-end and Boogie is the target IVL).

Proofs connecting an IVL with a back-end. Parthasarathy et al. [2024] show the soundness of the Viper back-end that translates to Boogie. In our work, we show that their back-end specific semantics respects our more generic version (§4.2). The work most closely related to the symbolic execution back-end presented in §4.1 is Zimmerman et al. [2024]'s formalization of a variant of Viper's symbolic execution back-end targeted at gradual verification. Due to their focus on gradual verification, they only target a simplified model of Viper that (unlike our symbolic execution) does not support fractional permission. As a consequence, they can use a simpler implementation that does not rely on continuation passing style and they can ignore some of the complexities described in §4.1 such as state consolidation. Also they formalize the symbolic execution via a derivation tree, while we implement it as an Isabelle/HOL function. Vogels et al. [2015] prove a formalization

of VeriFast's symbolic execution sound. Compared to our work, they do not have a semantics that captures different verification algorithms, or supports IDF or fractional permissions.

There is also work on non SL-based IVL back-end proofs. These back-ends typically have simple state models and use different algorithms compared to SL-based back-ends. For example, Parthasarathy et al. [2021] generate soundness proofs for Boogie's VCG, and Vogels et al. [2010] prove a VCG for a similar IVL sound once and for all. Garchery [2021] and Cohen and Johnson-Freyd [2024] validate certain logical transformations performed in the Why3 IVL verifier.

Angelic non-determinism. Angelic non-determinism [Floyd 1967] has been widely used from encoding partial programs [Bodík et al. 2010], to representing interaction between code written in multiple languages [Sammler et al. 2023; Guéneau et al. 2023], to encoding specifications [Floyd 1967; Song et al. 2023]. However, to the best of our knowledge, our work is the first to use angelic non-determinism to abstract over different verification algorithms. Vogels et al. [2015] and Song et al. [2023] both also use angelism for **exhale**, but do not abstract over or formally connect with diverse back-end algorithms, as we do. Instead, Vogels et al. [2015] use angelism to represent a symbolic execution algorithm, while Song et al. [2023] use angelism to encode the transfer of resources in a refinement calculus.

Implicit dynamic frames (IDF). IDF was originally presented with a fixed resource model (*i.e.*, full ownership to a heap location) and where the heap is represented as a *total* mapping from heap locations to values [Smans et al. 2012]. Parkinson and Summers [2012] formally showed the relationship between IDF and SL by defining a logic based on *total* heaps and separate permission masks that captures both. They also consider fixed resource models of IDF and SL (*i.e.*, fractional ownership to a heap location [Boyland 2003]). Our work generalizes the notion of a separation algebra [Calcagno et al. 2007; Dockins et al. 2009] to capture arbitrary resource models for IDF *and* SL in the same framework. In particular, the algebra does not fix a particular state representation. This enables, for instance, a *partial* heap instantiation for IDF that we use to formalize Viper's state model (§3.4). SteelCore [Swamy et al. 2020] is a framework with an extensible CSL to reason about concurrent F* [Swamy et al. 2016] programs. The extensibility of the framework is in particular demonstrated by allowing IDF-style preconditions of the restricted form $P * b$ (compared to the more general IDF assertions supported in our work), where P is an SL assertion, and b is a heap-dependent boolean expression framed by P (and similarly for postconditions).

Other approaches. In this paper, we showed how one can formally establish the soundness of translational SL verifiers, but there are also other approaches to building automated SL verifiers and establishing their soundness. Steel [Fromherz et al. 2021] is an SL-based proof-oriented programming language in F*. Steel programs are automatically proved correct using a type checker that is proved sound against SteelCore; the type checker uses an SMT solver to discharge proof obligations.

Keuchel et al. [2022] (building on the ideas of Vogels et al. [2015]) show how to build a verified symbolic execution based on a specification monad that allows expressing angelic and demonic non-determinism and assume and assert statements. They formalize (in Coq) two (structurally identical) versions of the symbolic execution algorithm: a deeply embedded version that allows execution and a shallow embedded version to prove soundness of the former. However, both versions represent the same algorithm; they do not consider different back-ends (like the verification condition generation back-end in §4.2).

Sammler et al. [2021] propose an approach to building sound verifiers that requires writing the verifier in a domain specific language called Lithium. Verifiers in Lithium can be automatically executed inside the Coq proof assistant and produce a foundational proof of correctness. Lithium-based verifiers are not translational, but work directly on the source-language program.

7 Conclusion

We have presented a formal framework for reasoning about the soundness of translational separation logic verifiers. We have defined an abstract IVL, whose state model can be instantiated with any IDF algebra. An operational and an equivalent axiomatic semantics allow one to connect the IVL to back-ends and front-ends, resp. Crucially, the semantics leverage dual non-determinism to capture different verification algorithms implemented by different back-end verifiers. We have illustrated the usefulness of our formal framework by instantiating it with elements of Viper, connecting it to two Viper back-ends, and using it to prove soundness of a front-end translation for an IDF-based concurrent separation logic. The main direction for future work is to use our formal framework to model additional IVLs and prove soundness of complex translational verifiers.

Acknowledgments

Data availability statement

All technical results presented in this paper have been formalized and proven in Isabelle/HOL, and our formalization is publicly available in our artifact [Dardinier et al. 2024b]. The development version is available at https://github.com/viperproject/viper-roots.

References

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 147, 30 pages. https://doi.org/10.1145/3360573

Sacha-Élie Ayoun, Xavier Denis, Petar Maksimovic, and Philippa Gardner. 2024. A hybrid approach to semi-automated Rust verification. *CoRR* abs/2403.15122 (2024). https://doi.org/10.48550/ARXIV.2403.15122 arXiv:2403.15122

Michael Backes, Catalin Hritcu, and Thorsten Tarrach. 2011. Automatically Verifying Typing Constraints for a Data Processing Language. In *Certified Programs and Proofs (CPP)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). https://doi.org/10.1007/978-3-642-25379-9_22

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005a. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO (Lecture Notes in Computer Science, Vol. 4111)*. Springer, 115–137. https://doi.org/10.1007/11804192_6

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005b. Symbolic Execution with Separation Logic. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3780)*, Kwangkeun Yi (Ed.). Springer, 52–68. https://doi.org/10.1007/11575467_5

Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Integrated Formal Methods (IFM)*, Nadia Polikarpova and Steve Schneider (Eds.). https://doi.org/10.1007/978-3-319-66845-1_7

Rastislav Bodík, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with angelic nondeterminism. In *POPL*. ACM, 339–352. https://doi.org/10.1145/1706299.1706339

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis (SAS)*, Radhia Cousot (Ed.). 55–72. https://doi.org/10.1007/3-540-44898-5_4

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *Logic in Computer Science (LICS)*. 366–375. https://doi.org/10.1109/LICS.2007.30

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/S10817-018-9457-5

Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 234–245. https://doi.org/10.1145/1993498.1993526

Joshua M. Cohen and Philip Johnson-Freyd. 2024. A Formalization of Core Why3 in Coq. *Proc. ACM Program. Lang.* 8, POPL, Article 60 (jan 2024), 30 pages. https://doi.org/10.1145/3632902

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

Thibault Dardinier, Peter Müller, and Alexander J. Summers. 2022a. Fractional resources in unbounded separation logic. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1066–1092. https://doi.org/10.1145/3563326

Thibault Dardinier, Gaurav Parthasarathy, and Peter Müller. 2023. Verification-Preserving Inlining in Automatic Separation Logic Verifiers. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 102 (apr 2023). https://doi.org/10.1145/3586054

Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. 2022b. Sound Automation of Magic Wands. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13372)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 130–151. https://doi.org/10.1007/978-3-031-13188-2_7

Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. 2024a. Formal Foundations for Translational Separation Logic Verifiers (extended version). *CoRR* abs/2407.20002 (2024). https://doi.org/10.48550/ARXIV.2407.20002

Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. 2024b. *Formal Foundations for Translational Separation Logic Verifiers − Artifact.* https://doi.org/10.5281/zenodo.13938950

Frank S. de Boer and Marcello M. Bonsangue. 2021. Symbolic execution formally explained. *Formal Aspects Comput.* 33, 4-5 (2021), 617–636. https://doi.org/10.1007/S00165-020-00527-Y

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *International Conference on Formal Engineering Methods (ICFEM)*, Adrián Riesco and Min Zhang (Eds.), Vol. 13478. 90–105. https://doi.org/10.1007/978-3-031-17244-1_6

Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A Fresh Look at Separation Algebras and Share Accounting. In *Asian Symposium on Programming Languages and Systems (APLAS)*, Zhenjiang Hu (Ed.). 161–177. https://doi.org/10.1007/978-3-642-10672-9_13

Marco Eilers, Malte Schwerhoff, and Peter Müller. 2024. Verification Algorithms for Automated Separation Logic Verifiers. *CoRR* abs/2405.10661 (2024). https://doi.org/10.48550/ARXIV.2405.10661

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *European Symposium on Programming (ESOP)*, Matthias Felleisen and Philippa Gardner (Eds.). https://doi.org/10.1007/978-3-642-37036-6_8

Robert W. Floyd. 1967. Nondeterministic Algorithms. *J. ACM* 14, 4 (1967), 636–644. https://doi.org/10.1145/321420.321422

Jean Fortin. 2013. *BSP-Why, un outil pour la vérification déductive de programmes BSP : machine-checked semantics and application to distributed state-space algorithms. (BSP-Why, a tool for deductive verification of BSP programs : sémantiques mécanisées et application aux algorithmes d'espace d'états distribués).* Ph. D. Dissertation. University of Paris-Est, France. https://tel.archives-ouvertes.fr/tel-00974977

Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: proof-oriented programming in a dependently typed concurrent separation logic. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473590

Quentin Garchery. 2021. A Framework for Proof-carrying Logical Transformations. In *Workshop on Proof eXchange for Theorem Proving (PxTP)*, Chantal Keller and Mathias Fleury (Eds.). https://doi.org/10.4204/EPTCS.336.2

Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 716–744.

Paolo Herms. 2013. *Certification of a Tool Chain for Deductive Program Verification. (Certification d'une chaine de vérification déductive de programmes).* Ph. D. Dissertation. University of Paris-Sud, Orsay, France. https://tel.archives-ouvertes.fr/tel-00789543

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods (LNCS, Vol. 6617)*. Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 2022. Verified symbolic execution with Kripke specification monads (and no meta-programming). *Proc. ACM Program. Lang.* 6, ICFP (2022), 194–224. https://doi.org/10.1145/3547628

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

Bernhard Kragl and Shaz Qadeer. 2021. The Civl Verifier. In *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 143–152. https://doi.org/10.34727/2021/ISBN.978-3-85448-046-4_23

K. Rustan M. Leino. 2008. This is Boogie 2. (2008). Available from http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, Edmund M. Clarke and Andrei Voronkov (Eds.). https://doi.org/10.1007/978-3-642-17511-4_20

K. Rustan M. Leino and Peter Müller. 2009. A Basis for Verifying Multi-threaded Programs. In *European Symposium on Programming (ESOP)*, Giuseppe Castagna (Ed.), Vol. 5502. Springer, 378–393. https://doi.org/10.1007/978-3-642-00590-9_27

Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Caroline Cronjäger, Petar Maksimovic, and Philippa Gardner. 2024. Compositional Symbolic Execution for Correctness and Incorrectness Reasoning. In *ECOOP (LIPIcs, Vol. 313)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:28. https://doi.org/10.4230/LIPICS.ECOOP.2024.25

Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021a. Gillian, Part II: Real-World Verification for JavaScript and C. In *CAV (2) (LNCS, Vol. 12760)*. Springer, 827–850. https://doi.org/10.1007/978-3-030-81688-9_38

Petar Maksimovic, José Fragoso Santos, Sacha-Élie Ayoun, and Philippa Gardner. 2021b. Gillian: A Multi-Language Platform for Unified Symbolic Analysis. *CoRR* abs/2105.14769 (2021). arXiv:2105.14769 https://arxiv.org/abs/2105.14769

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016a. Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 405–425. https://doi.org/10.1007/978-3-319-41528-4_22

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016b. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (Lecture Notes in Computer Science, Vol. 9583)*. Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3170)*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer, 49–67. https://doi.org/10.1007/978-3-540-28644-8_4

Matthew J. Parkinson and Alexander J. Summers. 2012. The Relationship Between Separation Logic and Implicit Dynamic Frames. *Logical Methods in Computer Science* 8, 3:01 (2012), 1–54. https://doi.org/10.2168/LMCS-8(3:1)2012

Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, and Alexander J. Summers. 2024. Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language. *Proc. ACM Program. Lang.* 8, PLDI, Article 208 (jun 2024), 25 pages. https://doi.org/10.1145/3656438

Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. 2021. Formally Validating a Practical Verification Condition Generator. In *Computer Aided Verification (CAV) (LNCS, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). 704–727. https://doi.org/10.1007/978-3-030-81688-9_33

Ingrid Rewitzky. 2003. Binary Multirelations. In *Theory and Applications of Relational Structures as Knowledge Instruments*. LNCS, Vol. 2929. Springer, 256–271. https://doi.org/10.1007/978-3-540-24615-2_12

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. *Logic in Computer Science (LICS)*, 55–74. https://doi.org/10.1109/lics.2002.1029817

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI*. ACM, 158–174. https://doi.org/10.1145/3453483.3454036

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL (2023), 775–805. https://doi.org/10.1145/3571220

José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part i: A Multi-language Platform for Symbolic Execution. In *PLDI*. ACM, 927–942. https://doi.org/10.1145/3385412.3386014

Malte Schwerhoff. 2016. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. Ph. D. Dissertation. ETH Zurich, Zürich, Switzerland. https://doi.org/10.3929/ETHZ-A-010835519

Malte Schwerhoff and Alexander J. Summers. 2015. Lightweight Support for Magic Wands in an Automatic Verifier (Artifact). *Dagstuhl Artifacts Ser.* 1, 1 (2015), 10:1–10:2. https://doi.org/10.4230/DARTS.1.1.10

Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.* 34, 1 (2012), 2:1–2:58. https://doi.org/10.1145/2160910.2160911

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL (2023), 1121–1151. https://doi.org/10.1145/3571232

Alexander J. Summers and Peter Müller. 2020. Automating deductive verification for weak-memory programs (extended version). *International Journal on Software Tools for Technology Transfer (STTT)* 22, 6 (2020), 709–728. https://doi.org/10.1007/S10009-020-00559-Y

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. https://doi.org/10.1145/2837614.2837655

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.* 4, ICFP (2020), 121:1–121:30. https://doi.org/10.1145/3409003

Viktor Vafeiadis. 2011. Concurrent Separation Logic and Operational Semantics. In *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 2011, Pittsburgh, PA, USA, May 25-28, 2011 (Electronic Notes in Theoretical Computer Science, Vol. 276)*, Michael W. Mislove and Joël Ouaknine (Eds.). Elsevier, 335–351. https://doi.org/10.1016/J.ENTCS.2011.09.029

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *Object Oriented Programming Systems Languages & Applications, (OOPSLA)*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). https://doi.org/10.1145/2509136.2509532

Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2009. A Machine Checked Soundness Proof for an Intermediate Verification Language. In *Theory and Practice of Computer Science, Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM) (Lecture Notes in Computer Science, Vol. 5404)*, Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia (Eds.). Springer, 570–581. https://doi.org/10.1007/978-3-540-95891-8_51

Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2010. A machine-checked soundness proof for an efficient verification condition generator. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung (Eds.). ACM, 2517–2522. https://doi.org/10.1145/1774088.1774610

Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2015. Featherweight VeriFast. *Log. Methods Comput. Sci.* 11, 3 (2015). https://doi.org/10.2168/LMCS-11(3:19)2015

Felix A. Wolf, Malte Schwerhoff, and Peter Müller. 2022. Concise outlines for a complex logic: a proof outline checker for TaDA. *Formal Methods in System Design* 61, 1 (2022), 110–136. https://doi.org/10.1007/S10703-023-00427-W

Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound Gradual Verification with Symbolic Execution. *Proc. ACM Program. Lang.* 8, POPL (2024), 2547–2576. https://doi.org/10.1145/3632927