Sound State Encodings in Translational Separation Logic Verifiers

HONGYI LING, ETH Zurich, Switzerland
THIBAULT DARDINIER, EPFL, Switzerland
ELLEN ARLT, MPI-SWS, Germany
PETER MÜLLER, ETH Zurich, Switzerland

Automated program verifiers are often organized into a front-end, which encodes an input program into an intermediate verification language (IVL), and a back-end, which proves that the IVL program is correct. Soundness of such translational verifiers requires that the back-end verification is sound and that correctness of the IVL program implies correctness of the input program. Existing formalizations for translational verifiers based on separation logic target the former, but support the latter only under the strong assumption that there exists a separation logic for the input program with the same state model as the IVL. This assumption is unrealistic in practice, especially since the state model also defines the supported separation logic resources.

We present the first formal framework for proving the soundness of translational separation logic verifiers with non-trivial state encodings. To be applicable to various front-ends and IVLs, our framework only assumes the existence of a homomorphic encoding relation between the front-end and IVL state models. At the core of our framework is a novel condition, backward satisfiability, which is crucial to guarantee the soundness of the front-end translation. We formalize our framework for front-end verifiers based on concurrent separation logic and separation logic IVLs, such as Raven, VeriFast, and Viper. We demonstrate its expressiveness by proving soundness for three common state encodings. Our framework and all proofs are formalized in Isabelle/HOL.

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

Additional Key Words and Phrases: Translational Verifier, Front-End Translation, Intermediate Verification Language, Implicit Dynamic Frames, State Encoding, Backward Satisfiability

ACM Reference Format:
Hongyi Ling, Thibault Dardinier, Ellen Arlt, and Peter Müller. 2026. Sound State Encodings in Translational Separation Logic Verifiers. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 330 (October 2026), 29 pages. https://doi.org/10.1145/3839462

1 Introduction

Many modern program verification tools are *translational verifiers*, *i.e.*, they split the verification into *front-ends*, which translate the input program with its specification into an *intermediate verification language* (IVL), and *back-ends*, which verify the resulting IVL program. This split enables the reuse of back-end verifiers across multiple front-ends, thus dramatically reducing the effort of developing verifiers. Examples of translational verifiers include Civl [25] and Dafny [28] based on the Boogie IVL [27], Gillian for C and Javascript [34] and Rust [2] based on GIL [17], Prusti [1], Nagini [14], and Gobra [59] based on Viper [35], as well as Frama-C [8] and Creusot [12] based on WhyML [15].

Authors' Contact Information: Hongyi Ling, ETH Zurich, Zurich, Switzerland, hongyi.ling@inf.ethz.ch; Thibault Dardinier, EPFL, Lausanne, Switzerland, thibault.dardinier@epfl.ch; Ellen Arlt, MPI-SWS, Kaiserslautern, Germany, earlt@mpi-sws.org; Peter Müller, ETH Zurich, Zurich, Switzerland, peter.mueller@inf.ethz.ch.

ACM 2475-1421/2026/10-ART330
https://doi.org/10.1145/3839462

```
def main(v: Int):
   x := alloc(f)
// { true }        { x.f ↦ _ }
   y := v    ||    x.f := v
// { y = v }       { ∃v'. x.f ↦ v' * v' = v }
   assert y = x.f
```

(a) Python-like front-end program, with concurrency and lazy field creation. The comments before and after the parallel composition represent the pre- and postconditions of the two threads, respectively. Here $x.f \mapsto \_$ represents the ownership of $x.f$ *without* any guarantee that the field `f` has been created yet, while $\exists v'.\, x.f \mapsto v'$ represents the ownership of $x.f$ *with* the guarantee that it has been created.

```
method main(v: Int) {
   // x := alloc(f)
   havoc x
   inhale x.f_added ↦ false

   // parallel composition
   exhale true
   exhale ∃b. x.f_added ↦ b * (b ⇒ x.f_IVL ↦ _)
   havoc y
   inhale y = v
   inhale ∃v'. x.f_added ↦ true * x.f_IVL ↦ v' * v' = v

   // assert y = x.f
   assert ∃v'. x.f_IVL ↦ v' * y = v'
}
```

```
method leftThread(v, y: Int) {
   inhale true
   y := v
   exhale y = v
}

method rightThread(v: Int, x: Ref) {
   inhale ∃b. x.f_added ↦ b * (b ⇒ x.f_IVL ↦ _)

   // x.f := v
   if (¬x.f_added)
      inhale x.f_IVL ↦ _
   x.f_added := true
   x.f_IVL := v

   exhale ∃v'. x.f_added ↦ true * x.f_IVL ↦ v' * v' = v
}
```

(b) Example translation of the front-end program into the Viper IVL, inspired by Nagini. The highlighted assertions represent the translations of the corresponding pre- and postconditions from Fig. 1a.

Fig. 1. An example front-end translation.

To support heap-manipulating and concurrent programs, many existing translational verifiers and their IVLs (GIL, Raven [19], VeriFast[1] [22], and Viper) are based on separation logic (SL) [43].

Soundness of a translational verifier requires (1) soundness of the back-end verifier and (2) soundness of the front-end translation, that is, that the successful verification of the IVL program implies correctness of the input program. While several existing works target back-end soundness for various IVLs [33, 38, 39, 7, 20], the state of the art does not offer practical techniques to prove soundness of realistic front-ends based on SL.

1.1 Challenge: Large Semantic Gap

The main challenge in establishing the soundness of a front-end translation into an SL-based IVL is to bridge the large semantic gap between the semantics and proof rules of the front-end programming language and the IVL. As an example, consider the program in Fig. 1a. This program allocates a new object with field `f`, then concurrently assigns the value of the argument `v` to local

[1]VeriFast does not have a human-readable IVL, but uses a uniform internal representation and back-end for multiple input languages (C, Java, Rust).

variable `y` and field `x.f`, before asserting that `y` and `x.f` are equal. Like in Python, the language used here adds the field `f` lazily to the object when it is first written: Attempting to read a field that has not yet been added results in a runtime error. To verify this program, a translation into the IVL must check that all fields that are read were previously assigned, in addition to ensuring the absence of memory errors and data races, and that the assertion at the end of method `main` holds.

Concurrency. Our IVL translation, inspired by the Viper-based Python verifier Nagini [14], is shown on Fig. 1b. Since the IVL does not support concurrency directly, we translate it using concurrent separation logic (CSL) [37] rules: The two front-end threads with their own pre- and post-conditions are translated into and verified as separate IVL methods `leftThread` and `rightThread`. In each of them, the translated precondition is added to the initially-empty IVL state via an **`inhale`** (also called *produce*) operation, the translated thread body is executed, and the translated postcondition is checked (and removed from the final state) via **`exhale`** (also called *consume*). In the main method, the concurrent part is abstracted to an exhale-havoc-inhale sequence that first removes the resources transferred to each thread via **`exhale`**s, then reflects the side effects of the threads on the local variable `y` by assigning a non-deterministic value (**`havoc`**), and finally adds the resources given back by each thread via **`inhale`**s.

Lazy field creation. As our target IVL has static field declarations (but does not support lazy field creation directly), our encoding needs to reflect for each field whether it has been added. To this end, the encoding represents each field f of the front-end program with two statically-declared fields in the IVL: the field f_{IVL} to store the value of the front-end field, and an additional boolean field f_{added} that indicates whether field f has been added to the object. As shown in the example: (1) Allocating the object x (as in the translation of `x := alloc(f)`) produces the SL resource $x.f_{added} \mapsto$ **`false`**, which provides the permission to add the field later, but does *not* generate any ownership to f_{IVL}. (2) Assigning to $x.f$ (as in the translation of `x.f := v` in the right thread) requires $x.f_{added} \mapsto$ **`false`** (if the field has not been added before) or $x.f_{IVL} \mapsto \_$ (if it has). In the former case, the resource $x.f_{IVL} \mapsto \_$ is produced lazily. In either case, $x.f_{IVL} \mapsto v$ is held after the assignment, thereby enabling subsequent reads. (3) Reading $x.f$ (as in the translation of **`assert`** `y = x.f`) requires $x.f_{IVL} \mapsto \_$, as usual.

This example demonstrates that the gap between front-end programs and their IVL encoding is substantial, affecting the state representation (here, an extra IVL field per front-end field), the operational execution (*e.g.*, concurrency in the front-end program vs. multiple sequential methods on the IVL), and proof rules (*e.g.*, CSL for the front-end vs. `inhale` and `exhale` primitives in the IVL). A soundness proof for a front-end translation needs to bridge this gap to show that verification of the IVL program implies the correctness of the front-end program.

1.2 State of the Art

As explained above, establishing the soundness of a front-end translation into an SL-based IVL, for example for the translation depicted in Fig. 1, requires bridging the large semantic gap between the semantics and proof rules of the front-end programming language and the IVL. Dardinier et al. [11] show that this gap can be reduced to proving that the correctness of the IVL encoding (with respect to an *axiomatic* semantics) implies the existence of a valid proof in a suitable *program logic* for the front-end programming language, as depicted in Fig. 2. This decomposition leverages the existence of sound separation logics for many advanced front-end features (*e.g.*, different kinds of concurrency [37, 18, 13, 57, 24], initialization semantics [42], non-local control flow [26], etc.) to prove the soundness of front-end translations. For example, Dardinier et al. show how to prove

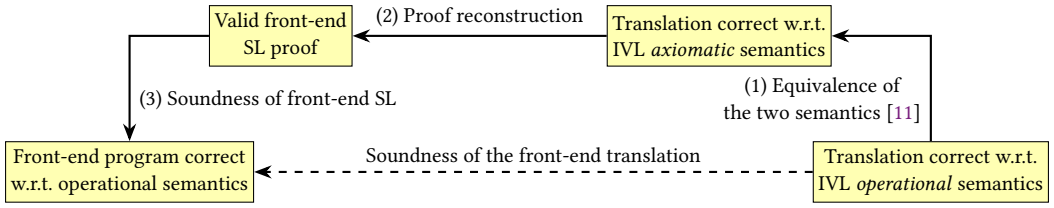

Fig. 2. Decomposing the soundness proof via a front-end SL. The dashed arrow represents the goal and the three solid arrows represent the decomposition. Existing work does the decomposition, but assumes that the two SLs have the same state model, which is unrealistic in practice. Our work lifts this key restriction, and enables full soundness proofs for front-ends with different state models to their IVLs.

the soundness of the concurrency translation from Fig. 1. However, their framework requires the front-end SL and the IVL to use the same state model, which is unrealistic in practice.

In practice, most front-ends use a state model that is vastly different from the fixed state model of the IVL. One major discrepancy is how memory and data are represented. For example, C's memory regions or Java's arrays might be encoded into an object-based memory model in an IVL, which does not natively support a notion of consecutive addresses. Moreover, our example from Fig. 1a encodes each Python field by a pair of fields to track which fields have been added to an object.

Important discrepancies also arise for the information tracked as part of the verification state, such as the separation logic resources held in a state. For example, front-end states may include designated resources to execute a method call [21] or perform an I/O operation [40, 46], which need to be encoded into standard SL predicates in the IVL. Moreover, front-end and IVL may use a different permission model. For example, Viper [35], VeriFast [22], and Raven[2] [19] all build in fractional permissions [6, 5] but have front-ends that use binary[3] permissions (*e.g.*, the encoding of I/O tokens in VeriFast [40] or Viper [46, 41]), counting permissions [5] (*e.g.*, in Gobra [59] and the encoding of relaxed separation logic [57] in Viper [47, 35]), or duplicable permissions (*e.g.*, duplicable invariants over immutable shared data [42]).

Due to these discrepancies, none of these encodings can be proven sound by the existing framework by Dardinier et al.

1.3 This Work

We develop the first formal framework for proving the soundness of front-end translations that use different state models from their IVLs, thus lifting the key restriction of existing work and enabling soundness proofs for practical front-end translations. To be compatible with a wide range of front-end separation logics and IVLs, our framework simply assumes the existence of a homomorphic encoding relation between front-end and IVL states. We identify a novel condition on this encoding relation, which we call *backward satisfiability*, that is crucial to guarantee the soundness of the front-end translation. To show the practicality of our approach, we develop a formal framework to prove the soundness of front-end translations based on concurrent separation logic (CSL) [37], and instantiate it with three different front-end translations inspired by existing Viper front-ends, including the one in Fig. 1. We also show that backward satisfiability is essential for soundness, by uncovering several subtle sources of unsoundness in state encodings that affect

[2]Raven employs user-definable resource algebras for its ghost resources, but still uses fractional permissions for its regular program states.
[3]That is, one either owns a full permission or no permission at all.

existing verifiers, but have not been described before. More broadly, we show that violations of our framework's requirements underlie several previously-reported soundness bugs in verifiers such as Prusti, Nagini, and VerCors [4].

In addition to proving the soundness of existing front-ends, our framework also guides the development of new sound front-end translations. Developing a front-end typically involves three steps: choosing a suitable IVL, designing the state encoding relation between front-end and IVL states, and defining the translations of front-end expressions, assertions, and statements. Our framework's requirements support each of these steps in two ways. First, they provide design guidance even before any proof is attempted: because a sound design must satisfy them, they rule out large families of unsound choices and narrow the design space. Second, they enable incremental verification: once a design decision is made, the corresponding requirement is a self-contained proof obligation that can be discharged immediately, allowing to catch an unsound choice early rather than after the whole translation is built.

In summary, we make the following technical contributions:

(1) We identify a set of sufficient conditions, including the novel *backward satisfiability*, that guarantee the soundness of state encodings (§2 and §3).
(2) We identify several subtle sources of unsoundness in state encodings that affect existing verifiers, but have not been described before (§2).
(3) We develop a formal framework for proving the soundness of front-end translations. It is parametric in the front-end state model, expressions, assertions, statements, and logic, as well as their translations into the IVL. It is applicable to the translations used in practical SL-verifiers for both sequential and concurrent programs (§3).
(4) We demonstrate the expressiveness of our framework by instantiating it for three practical front-end translations inspired by existing verifiers (§4).
(5) The framework including all proofs is formalized in Isabelle/HOL.

Our framework, including all formalizations and proofs, is mechanized in Isabelle/HOL [36] and the mechanization is publicly available [31].

2 Overview: Homomorphism, Spurious Splittability, and Backward Satisfiability

In this section, we present key properties that a front-end translation must satisfy to be sound. We assume the existence of a state encoding relation between front-end and IVL states, which is general enough to capture many practical front-end translations (§2.1). We explain why this relation should be homomorphic, and show how violating this requirement can lead to unsoundness (§2.2). We then describe the key problem of *spurious splittability*, where the IVL state resulting from encoding a front-end state is split into two IVL states that have no front-end counterparts (§2.3). We explain how spurious splittability may lead to unsoundness and introduce our solution, *backward satisfiability*, to prevent this (§2.4). Finally, we show examples where spurious splittability, and the absence of backward satisfiability, lead to unsoundness (§2.5).

2.1 State Encoding Relations

Discrepancies between front-end and IVL states can stem from many different sources: a different heap structure (*e.g.*, in Fig. 1), a different permission model (*e.g.*, binary vs. fractional permissions), or even a different representation of the same permission model (*e.g.*, rational vs. real numbers for fractional permissions). To abstract over these different aspects and make our approach generically applicable to different front-ends and IVLs, our framework is parametric in the front-end and IVL state models, and we simply assume the existence of a state encoding relation $\cdot \sim_S \cdot$ between them. Intuitively, $\omega \sim_S \Omega$ holds when the IVL state Ω represents the front-end state ω.

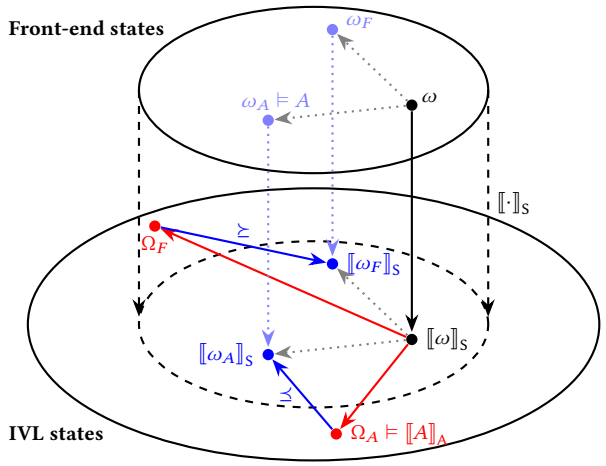

Fig. 3. An illustration of our setting (in black), the problem of spurious splittability (in red), and the backward satisfiability solution (in blue). The black dashed arrows represent the state encoding function $[\![\cdot]\!]_S$ from front-end states to IVL states, and the black dashed ellipse represents the image of $[\![\cdot]\!]_S$. The red arrows represent a possible spurious split of the IVL state $[\![\omega]\!]_S$ into Ω_A (that satisfies the translated assertion $[\![A]\!]_A$) and Ω_F (the frame), that have no front-end counterparts. The blue arrows show how the backward satisfiability of A guarantees the existence of front-end states ω_A and ω_F, such that ω_A satisfies A, $[\![\omega_A]\!]_S$ is smaller or equal to Ω_A, and $[\![\omega_F]\!]_S$ is larger or equal to Ω_F.

Relations from encoding functions. The state encoding *relation* is often derived naturally from a *function* $[\![\cdot]\!]_S$ that encodes a front-end separation logic state into the corresponding IVL state, in which case $\omega \sim_S \Omega$ is defined as $\Omega = [\![\omega]\!]_S$. For example, $[\![\cdot]\!]_S$ could encode binary permissions into fractional permissions by setting a permission of 1 for locations owned by the state, and 0 for the others. In the example from Fig. 1, our function $[\![\cdot]\!]_S$ encodes states as follows: The ownership of a non-existent field `f` of a created object `x` is encoded as the ownership of f_{added} of `x` with its value set to **`false`**; the ownership of a created field `f` with value v is encoded as the ownership of both f_{added} and f_{IVL}, where the former has value **`true`** and the latter stores the actual value v. The situation is represented in black in Fig. 3: Our function $[\![\cdot]\!]_S$ encodes the front-end states into (typically) a subset of the IVL states. For example, the image of the encoding function from binary permissions is the subset of IVL states with permissions 0 or 1.

The need for a relation. State encoding functions are too restrictive to capture some front-end encodings, for instance, translations that encode duplicable permissions into fractional permissions, as no fixed positive fraction, however small, can be duplicated. However, duplicable permissions can be interpreted in the IVL as *any arbitrary* positive fraction: A front-end state with a single duplicable resource then corresponds to infinitely many IVL states, each with a different fraction. Such encodings are easily captured by our binary relation $\cdot \sim_S \cdot$, as we will show in §4.

2.2 The Homomorphism Requirement

At the core of separation logic is the concept of *separation*: A state ω satisfies the assertion $P * Q$ if it can be split into two states ω_P and ω_Q such that ω_P satisfies P, ω_Q satisfies Q, and $\omega = \omega_P \oplus \omega_Q$, where $\oplus$ is a partial commutative and associative binary operation. In particular, both front-end and IVL states are equipped with their operator $\oplus$. For the translation to be meaningful, we require the state encoding relation to be homomorphic with respect to $\oplus$.

```
method m(x: Ref, y: Ref, z: Ref)
  requires x.f ↦[ϵ] _ * y.f ↦[ϵ] _ * z.f ↦[ϵ] _
{
  if (x = y ∧ x = z)
    assert false
}
```

```
method m(x: Ref, y: Ref, z: Ref) {
  inhale x.f ↦[1/2] _ * y.f ↦[1/2] _ * z.f ↦[1/2] _
  if (x = y ∧ x = z)
    assert false
}
```

Fig. 4. Encoding counting permissions $k \cdot \epsilon$ as fractions $1 - \frac{1}{2^k}$ is unsound. The front-end program on the left permits the three inputs to be aliased, but the IVL program on the right does not.

For simplicity, we describe here the homomorphism requirement for the case where $\cdot \sim_S \cdot$ is defined by an encoding function $[\![\cdot]\!]_S$, but we adapt it later to the general case of a relation. The encoding function $[\![\cdot]\!]_S$ is required to be homomorphic: For all front-end states ω_1 and ω_2, $[\![\omega_1 \oplus \omega_2]\!]_S = [\![\omega_1]\!]_S \oplus [\![\omega_2]\!]_S$. This is necessary so that the separating conjunction ($*$) in the IVL program (*e.g.*, when adding or removing resources via inhale or exhale) corresponds to the separating conjunction in the front-end program (*e.g.*, in its parallel rule).

Unsoundness without homomorphism. To see why the homomorphism requirement is necessary, consider encoding a front-end state model with counting permissions into an IVL state model with fractional permissions. Counting permissions [5] is a permission model that allows one to subtract from a full permission an unbounded number of units. In this model, a state holds either $1 - k \cdot \epsilon$ or $k \cdot \epsilon$ permission to a heap location (for any non-negative k), where a full permission 1 is required to write to a location, and any positive number of units $k \cdot \epsilon$ provides read access. Counting permissions are, for instance, used in the Chalice verifier [29].

A simple (but unsound) encoding of counting permissions into an IVL offering fractional permissions [6], such as VeriFast and Viper, is to encode $k \cdot \epsilon$ as the fraction $1 - \frac{1}{2^k}$, and $1 - k \cdot \epsilon$ as the complementary fraction $\frac{1}{2^k}$. This encoding ensures that the sum of the fractions for $1 - k \cdot \epsilon$ and $k \cdot \epsilon$ provides write permission and, for positive k, each fraction is positive and, thus, provides read permission. However, this encoding is *not* homomorphic: Let ω^k be a front-end state with $k \cdot \epsilon$ permission to some heap location `x.f`, and Ω^p be an IVL state with a fractional permission p to the corresponding heap location. Then $[\![\omega^1 \oplus \omega^1]\!]_S = [\![\omega^2]\!]_S = \Omega^{1-\frac{1}{2^2}} = \Omega^{\frac{3}{4}}$, whereas $[\![\omega^1]\!]_S \oplus [\![\omega^1]\!]_S = \Omega^{1-\frac{1}{2^1}} \oplus \Omega^{1-\frac{1}{2}} = \Omega^1 \neq \Omega^{\frac{3}{4}}$. In other words, the encoding of the sum of the front-end states $\omega^1 \oplus \omega^1$ has *strictly less* permission than the sum of the individual encodings.

This violation of the homomorphic requirement results in an unsound front-end translation, as illustrated by the example in Fig. 4. The front-end program on the left might lead to an assertion violation because the counting permissions required in the precondition do *not* prevent all three inputs from being aliased, as there is no bound on the number of units for each location. In contrast, the assertion inhaled by the IVL program (on the right) rules out the case that all three inputs are aliased since it is not possible to own more than one full permission to each location. Therefore, the IVL program verifies even though the input program should not.

Note that this unsoundness is independent of the particular fraction chosen to represent one unit, as long as this fraction is strictly positive; smaller fractions just need more aliases to exhibit the unsoundness. Summers and Müller [47] use an alternative encoding, in which a unit is encoded as a *symbolic* fraction, together with the axioms $f > 0$ and $\forall k \in \mathbb{N}.\, k \cdot f < 1$. This representation makes it harder for an SMT-based verifier to *exploit* the unsoundness, but it is still present as there is no fraction f that satisfies these axioms.

```
method main() {
  x := alloc(f)
  fork reader(x)
  y := x.f
}

method reader(x: Ref)
  requires x.f ↦ _
{
  var v := x.f
  ...
}
```

```
method main() {
  havoc x
  inhale x.f ↦ _
  exhale ∃p > 0. x.f ↦[p] _
  y := x.f
}

method reader(x: Ref) {
  inhale ∃p > 0. x.f ↦[p] _
  var v := x.f
  ...
}
```

Fig. 5. Encoding binary permissions using fractional permissions is generally unsound. The IVL program on the right allows reading `x.f` after the fork because of the use of fractional permissions), and thus is correct. In contrast, the front-end program uses binary permissions, and thus should not be verified.

2.3 Spurious Splittability

As discussed in §1, front-end translations rely on SL operations to add resources to the verification state (*produce*) and to remove them (*consume*). IVLs provide these operations (*e.g.*, **`inhale`** and **`exhale`** in Viper) to encode SL proof rules. For example, the front-end translation from Fig. 1 uses these operations to model allocation and parallel composition, which are not supported by the IVL.

In general, a front-end statement C (*e.g.*, `x := alloc(f)`) can be modeled in the IVL via the corresponding rule from the front-end separation logic: If the triple $\{P\}\ C\ \{Q\}$ is valid in the front-end separation logic, then a possible (and typical) translation for C in the IVL is **`exhale`** `P;` **`inhale`** `Q` (assuming that C does not modify local variables for simplicity). Starting in the IVL state Ω, **`exhale`** `P` will split Ω into $\Omega_P \oplus \Omega_F$ (where Ω_P satisfies P) and remove Ω_P. The resulting state Ω_F represents the frame, *i.e.*, the resources that are unaffected by the execution of C. The **`inhale`** `Q` statement will then add another state Ω_Q to the frame Ω_F, resulting in the state $\Omega_Q \oplus \Omega_F$. The state Ω_Q, which satisfies Q, models how the statement C modified Ω_P. Intuitively, this encoding is justified by the frame rule of the front-end separation logic, with Ω_F used as frame.

Unfortunately, this intuitive justification might not be valid, because the split $\Omega_P \oplus \Omega_F$ might be *spurious*: As represented in Fig. 3 (in red), the IVL states Ω_P and Ω_F might not have front-end counterparts, even if the initial IVL state Ω has a front-end counterpart (*i.e.*, a front-end state ω such that $\Omega = [\![\omega]\!]_S$). In such a case, the frame rule of the front-end separation logic cannot be used soundly with the IVL state Ω_F, as it has only been proven sound for front-end states. Such spurious splits arise because the front-end states generally correspond to a *subset* of the IVL states; in other words, there are generally infinitely many more IVL states than front-end states, and thus infinitely many IVL states without front-end counterparts. We call IVL states that have a front-end counterpart *concrete* states, and *spurious* those that do not.

Unsoundness from spurious splits. As a simplified example (we show more examples below), consider a front-end that uses binary permissions. The encoding into an IVL with fractional permissions attempts to improve framing by determining with a simple pre-analysis whether a method updates a heap location. If not, it encodes permission to that location as an arbitrary positive fraction rather than full permission, such that callers can frame the value around calls. In this setting, the front-end program on the left of Fig. 5 does not verify because there is no permission to access `x.f` after the fork. However, the encoding on the right verifies because the

assertion $\exists p > 0.\, \texttt{x.f} \overset{p}{\mapsto} \_$ in the IVL program (on the right of Fig. 5) can be satisfied by any positive fraction, enabling the read `y := x.f` in the main thread. This unsoundness is caused by spurious splittability: an IVL state with full permission to `x.f` can be split into two states, each with 1/2-permission (one being passed to the forked thread and the other retained to read `x.f` subsequently). However, these states have no corresponding front-end states, such that the same reasoning steps are not valid there. We show in §2.5 that similar problems arise even with an assertion syntax suitably restricted for the front-end state model.

2.4 Backward Satisfiability

The unsoundness in the previous example comes from the fact that the encoded precondition $\exists p > 0.\, \texttt{x.f} \overset{p}{\mapsto} \_$ can be satisfied by a spurious IVL state, which results in a spurious frame. In particular, no smaller concrete state satisfies the encoded precondition, such that *any* split that enables verifying the whole IVL program is spurious.

We prevent this source of unsoundness by requiring front-end assertions to satisfy a novel soundness condition. Intuitively, a front-end assertion A is *backward satisfiable* if for any[4] IVL state Ω_A that satisfies its translation $[\![A]\!]_{\mathrm{A}}$, there exists a front-end state ω_A that satisfies A and whose encoding $[\![\omega_A]\!]_{\mathrm{S}}$ is smaller than Ω_A. Intuitively, and as depicted in Fig. 3 in blue, backward satisfiability guarantees the existence of an alternative, non-spurious split $[\![\omega]\!]_{\mathrm{S}} = [\![\omega_A]\!]_{\mathrm{S}} \oplus [\![\omega_F]\!]_{\mathrm{S}}$ for some front-end states ω_A (satisfying A) and ω_F, such that $[\![\omega_F]\!]_{\mathrm{S}} \succeq \Omega_F$. The latter ensures that the IVL frame Ω_F can be used soundly with the frame rule from the front-end SL.[5]

Importantly, backward satisfiability holds for many basic SL assertions, such as points-to $\texttt{x.f} \overset{p}{\mapsto} v$ and pure assertions (*i.e.*, without permissions). Backward satisfiability is preserved by many important SL connectives. For example, if A and B are backward satisfiable, then so is $A * B$ (with the straightforward definition $[\![A * B]\!]_{\mathrm{A}} \triangleq [\![A]\!]_{\mathrm{A}} * [\![B]\!]_{\mathrm{A}}$). Unfortunately, backward satisfiability is *not* preserved by all connectives, such as magic wands (separating implications), as we show next.

2.5 Unsoundness without Backward Satisfiability

In this subsection, we show two examples that violate backward satisfiability because of the magic wand connective, and how the soundness of front-end translations can be compromised when the backward satisfiability condition is violated. The framework presented in this paper allows one to uncover such subtle sources of unsoundness and to prove that correct encodings are indeed sound.

Encoding binary permissions into fractional permissions. The example in Fig. 6 encodes the input program on the left into the identical IVL program on the right. The front-end program on the left uses binary permissions, whereas the IVL encoding on the right uses fractional permissions. The assertion W is defined using a magic wand (separating implication) connective:

$$W \triangleq \texttt{x.f} \mapsto \_ \mathrel{-\!\!*} (\texttt{x.f} \mapsto \_ * \texttt{y.g} \mapsto \_)$$

Intuitively, it expresses that one can give up the wand as well as ownership of `x.f` to obtain ownership of both `x.f` and `y.g` (that is, apply modus ponens).

Magic wands themselves represent resources (the wand's *footprint*) that, combined with the left-hand side of the wand, entail the right-hand side. With binary permissions, W's footprint contains either permission to `x.f` (so that combining it with the left-hand side yields false, thereby trivially satisfying the right-hand side) or permission to `y.g`. No matter which footprint is chosen, the postcondition represents more than full permission to a heap location and is thus unsatisfiable.

[4]More precisely, backward satisfiability only considers IVL states Ω_A that are smaller than the encoding $[\![\omega]\!]_{\mathrm{S}}$ of some front-end state ω, *i.e.*, such that $[\![\omega]\!]_{\mathrm{S}} = \Omega_A \oplus \Omega_R$ for some IVL state Ω_R.
[5]Throughout the paper, we assume an affine front-end SL.

```
method m(x: Ref, y: Ref)
  requires x.f ↦ _
  ensures W ∗ W ∗ W
{ skip }
```

```
method m(x: Ref, y: Ref) {
  inhale x.f ↦ _
  skip
  exhale W ∗ W ∗ W
}
```

Fig. 6. Encoding binary permissions using fractional permissions is generally unsound. The IVL program on the right verifies because there are fractional permissions for the footprints of the three magic wands W. With the binary permissions used by the front-end program on the left, it is not possible to provide sufficient permissions for all three wands.

Hence, verification should fail. Nevertheless, verification of the IVL program succeeds. With fractional permissions, one can choose W's footprint as $\texttt{x.f} \overset{1/3}{\mapsto} \_$, such that the right-hand side is entailed trivially. With this choice, the postcondition holds, and verification succeeds.

This unsoundness stems from the violation of backward satisfiability by W (despite the fact that both sides of the magic wand are backward satisfiable): the IVL may satisfy the translation of W in a state Ω_A with $\frac{1}{3}$ permission to `x.f`. However, the *only* front-end state ω_A that satisfies W has full permission to `x.f` and, thus, its encoding is *not* smaller than or equal to Ω_A.

This example also demonstrates that a commonly-used state encoding is sound only if it is combined with appropriate restrictions on the allowed assertions. For instance, disallowing binary permissions inside magic wands would rule out our unsound example.

Encoding rational permission amounts into real permission amounts. Our next example shows that the common practice of encoding SLs with rational permission amounts into automatic verifiers targeting real permission amounts is surprisingly unsound in the presence of the magic wand connective. Consider the following two assertions:

$$P_1 \triangleq \left(\exists v.\, \texttt{y.g} \mapsto v * 0 < v * v^2 \leq \frac{1}{2}\right) \mathrel{-\!\!*} \left(\exists v.\, \texttt{y.g} \mapsto v * \texttt{x.f} \overset{v}{\mapsto} \_\right)$$

$$P_2 \triangleq \left(\exists v.\, \texttt{y.g} \mapsto v * 0 < v < 1 * (1-v)^2 \geq \frac{1}{2}\right) \mathrel{-\!\!*} \left(\exists v.\, \texttt{y.g} \mapsto v * \texttt{x.f} \overset{v}{\mapsto} \_\right)$$

In both of them, the left-hand side of the magic wand constrains the value v, which is used as a permission amount for `x.f` on the right-hand side. The predicate $\texttt{y.g} \mapsto v$ enforces the values v on both sides to be the same. P_1 constrains v to be in the half-open interval $(0, \frac{1}{\sqrt{2}}]$, whereas P_2 allows values for v in $(0, 1 - \frac{1}{\sqrt{2}}]$. In both wands, the left-hand side provides no permission for `x.f`, so in both cases, $\texttt{x.f} \overset{v}{\mapsto} \_$ is part of the footprint of the wand.

```
method m(x: Ref, y: Ref)
  requires x.f ↦ _
  ensures P1 ∗ P2
{ skip }
```

```
method m(x: Ref, y: Ref) {
  inhale x.f ↦ _
  skip
  exhale P1 ∗ P2
}
```

Fig. 7. Encoding rational permission amounts as real numbers is unsound. The front-end program on the left and the IVL program on the right interpret the permission amounts in the postcondition differently, such that the IVL program verifies successfully even though the front-end program should not.

The front-end and IVL methods in Fig. 7 require full ownership to `x.f` and ensure the separating conjunction $P_1 * P_2$. The two methods look identical, but use rationals and reals as permission

amounts, respectively. The footprints of both magic wands need to contain $\texttt{x.f} \overset{v}{\mapsto} \_$ for *every* value v permitted by the left-hand side. The IVL program, which uses real numbers as permission amounts, thus, uses $\frac{1}{\sqrt{2}}$ for P_1 and $1 - \frac{1}{\sqrt{2}}$ for P_2. In real arithmetic, these add up to a full permission, which is provided by the precondition. Hence, verification of the IVL program succeeds. However, the front-end logic, which uses rational permission amounts, cannot represent the amounts $\frac{1}{\sqrt{2}}$ and $1 - \frac{1}{\sqrt{2}}$ precisely and, thus, needs to conservatively over-approximate them. As a result, their sum exceeds the full permission provided in the precondition, which causes verification to fail.

The unsoundness in this example again stems from the violation of backward satisfiability, by both P_1 and P_2 (despite the fact that all sides of the magic wands are backward satisfiable): the IVL may satisfy P_1 (resp. P_2) in a state Ω_A with $\frac{1}{\sqrt{2}}$ (resp. $1 - \frac{1}{\sqrt{2}}$) permission to $\texttt{x.f}$, while the front-end must provide strictly more amount of permission for ω_A to satisfy P_1 (resp. P_2).

This example shows that encoding rational permission amounts as reals is generally unsound. Since it should in principle be possible to formalize fractional permissions with real numbers instead of rational ones, we do not expect this unsoundness to compromise actual verification results. However, the mismatch makes it impossible to prove a formal soundness result for front-ends using this common state model encoding.

These unsoundnesses stem from the mismatch between the front-end and IVL state models: when the two coincide, the state encoding can be taken to be the identity, backward satisfiability holds trivially, and *all* connectives (including magic wands and universal quantifications) can be translated soundly. Conversely, for front-ends with a sufficiently expressive assertion language, backward satisfiability is necessary: whenever an assertion's translation violates it *and* the front-end can express the corresponding upper bound and frame (as in all our examples), one can construct an unsound method as above, in which the IVL translation verifies but the front-end program does not.

3 A Framework for Proving Front-End Translations Sound

Our core contribution is a formal framework that allows one to prove that the state model encoding of a front-end translation is sound. In this section, we formalize our framework: the semantic model of the IVL (§3.1), its parameters (§3.2) and their soundness requirements (§3.3), our soundness result (§3.4), and a generalization in which the state encoding is a relation rather than a function (§3.5). All results are formalized in Isabelle/HOL. Although the formalization uses Viper as the IVL, all requirements and results are independent of any specific IVL, so we present them for a general SL-based IVL.

3.1 Preliminaries

We adopt much of our semantic model of the IVL from Dardinier et al. [11].

IDF algebras. To make our framework widely applicable, our formalization captures both IVLs based on standard SL (*e.g.*, VeriFast) and IVLs based on implicit dynamic frames (IDF) [44, 45] (*e.g.*, Viper). To achieve that, we formalize states as IDF algebras, a generalization of separation algebras that also captures IDF state models:[6]

Definition 1 (IDF algebra)**.** An IDF algebra is a tuple $(\Sigma, \oplus, |\cdot|, \mathsf{stable}, \mathsf{stabilize}, \mathsf{unit})$ that satisfies a set of axioms (listed in the extended version [32, §A]), where Σ is a set of states, $\oplus : \Sigma \times \Sigma \rightharpoonup \Sigma$ is

[6] Compared with the definition from Dardinier et al. [11], we make the unit function explicit and independent from stabilize and $|\cdot|$; the definition from Dardinier et al. [11] can be derived from ours by setting $\mathsf{unit}(s) \triangleq \mathsf{stabilize}(|s|)$. Our generalization is crucial to capture front-end state models where $\mathsf{stabilize}(|\cdot|)$ does not satisfy the properties for $\mathsf{unit}(\cdot)$, *e.g.*, the immutable heaps from §4.4, where $\mathsf{stabilize}(|h|) = h$ but $\mathsf{unit}(h) = \lambda l.\ \bot$.

a partial, commutative, and associative addition on Σ, $|\cdot|$, stabilize, and unit are endomorphisms of Σ, and stable is a predicate on Σ.

Intuitively, $\oplus$ denotes the composition of states, $|\cdot|$ projects a state on its largest duplicable part, stable decides the self-containedness of a state (*i.e.*, the state holds enough ownership to all its fields so that none of them may be modified by the environment), stabilize projects a state on its largest stable part, and unit denotes the unit element of $\oplus$.

IDF assertions separate ownership information from value constraints. For instance, SL's points-to assertion $\texttt{x.f} \mapsto \texttt{v}$ is expressed in IDF as `acc(x.f) * x.f = v`. To ensure sound framing, IDF requires assertions to be self-framing; that is, an assertion may constrain a heap location only if it includes the permission to that location. Consequently, `acc(x.f) * x.f = v` is self-framing, but `x.f = v` alone is not. We defined self-framedness of assertions and other related concepts as follows:

Definition 2. Let P be an IDF assertion (*i.e.*, a set of states from an IDF algebra).

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

IVL semantics. An IVL state consists of a store of local variables and a customized heap $\varphi \in \Sigma$ equipped with an IDF algebra structure, *i.e.*, the set of IVL states $\Sigma_V = (\mathsf{Var} \rightharpoonup V) \times \Sigma$, where V is the set of values and the store $\mathsf{Var} \rightharpoonup V$ is instantiated to the agreement algebra, *i.e.*, addition on stores is defined only for identical stores and yields the argument store, $|\cdot|$, stabilize, and unit are instantiated to the identity function, and $\mathsf{stable}(s)$ holds for any store s.

We require that the IVL supports three primitive statements **`havoc`** x, **`inhale`** A, **`exhale`** A, and sequential compositions $C_1; C_2$ and conditional statements **`if`**(b) $\{C_1\}$ **`else`** $\{C_2\}$ in order to translate some front-end control structures. Intuitively, **`havoc`** x erases the value of local variable x by overwriting it non-deterministically, **`inhale`** A adds the resource specified by assertion A into the current state and assumes the value constraints in A, and **`exhale`** A checks that A holds in the current state and subsequently removes the resource corresponding to A from the current state.

We assume an axiomatic semantics for the IVL. The axiomatic semantic judgements are of the form $\Delta \vdash [P]\ C\ [Q]$, where Δ is a type context[7], P and Q are IVL semantic assertions (*i.e.*, sets of IVL states), and C is an IVL statement. The axiomatic semantic rules of the aforementioned statements are given in the extended version [32, §A].

3.2 Framework Parameters

To capture a wide range of front-ends, our framework offers numerous parameters that describe the input language to the front-end as well as the translation performed by the front-end.

Parameters describing the input language. The framework parameters for the input language describe, for instance, the available boolean expressions, assertions, and statements. Fig. 8 provides an overview of the parameters; in the following, we explain some of the most interesting ones; further details are included in the extended version [32].

[7]We omit typing details here, but our Isabelle/HOL formalization includes them: it requires the state encoding to preserve well-typedness and deals only with well-typed states (whose store and heap contain values of the types prescribed by Δ). All states in §3 and §4 are implicitly assumed well-typed.

Framework parameter	Description
T_F	Set of front-end types.
V_F	Set of front-end values.
$\mathrm{tp} : V_F \to T_F$	Function that assigns each value $v \in V_F$ a type $t \in T_F$.
H_F	Set of front-end *ghost heaps* used for reasoning about the resources in front-end states. They must be instantiated as an IDF algebra.
H_P	Set of front-end *program heaps*.
$\mathrm{embed}_H : H_P \to H_F$	Function that embeds program heaps onto ghost heaps.
BE_F	Set of front-end boolean expressions.
$I_\mathrm{B} : BE_F \to (\Sigma_F \to \{\texttt{true}, \texttt{false}\})$	Interpretation for each boolean expression $b \in BE_F$ as a partial predicate on front-end ghost states.
$\mathrm{fv}_\mathrm{B} : BE_F \to \mathcal{P}(\mathrm{Var})$	(Possibly overapproximating) finite free variable set of a boolean expression.
A_{Fp}	Set of input assertions.
$\mathrm{wf}_A^{\Delta_F}(\cdot)$	Well-formedness predicate on input assertions in a given type context Δ_F.
$I_\mathrm{A}^{\Delta_F} : A_{Fp} \to \mathcal{P}(\Sigma_F)$	Interpretation of each input assertion $A_p \in A_{Fp}$ as a set of front-end ghost states satisfying the assertion under type context Δ_F.
$\mathrm{fv}_\mathrm{A} : A_{Fp} \to \mathcal{P}(\mathrm{Var})$	(Possibly overapproximating) finite free variable set of an input assertion.
C_{Fp}	Set of primitive statements.
$\mathrm{wf}_C^{\Delta_F}(\cdot)$	Well-formedness predicate on primitive statements in a given type context Δ_F.
Rules of the form $\sigma \xrightarrow{c}_{\Delta_F} \sigma'$	Operational semantic reduction rules for each primitive statement $c \in C_{Fp}$. $\sigma, \sigma' \in \Sigma_P$ are program pre- and post-states, and Δ_F is a type context.
Rules of the form $\Delta_F \vdash_p \{P\}\, c\, \{Q\}$	CSL rules for primitive statements $c \in C_{Fp}$. Δ_F is a front-end type context, and P and Q are front-end semantic assertions (*i.e.*, sets of front-end ghost states).
$\mathrm{wr} : C_{Fp} \to \mathcal{P}(\mathrm{Var})$	(Possibly overapproximating) finite modified variable set of a primitive statement.
$\mathrm{fv}_\mathrm{C} : C_{Fp} \to \mathcal{P}(\mathrm{Var})$	(Possibly overapproximating) finite free variable set of a primitive statement.

Fig. 8. Framework parameters for the input language.

Types and boolean expressions are fully described by the respective framework parameters (T_F and BE_F). The parameter A_{Fp} for assertions describes a set of so-called *input assertions*; the assertions supported by the front-end are then obtained using the following grammar, where $A_p \in A_{Fp}$, $b \in BE_F$, $t \in T_F$, and P is a designated symbol for recursive occurrences (as we explain below):

$$A ::= \; A_p \mid b \mid A * A \mid A \vee A \mid b \Rightarrow A \mid \exists x : t.\, A \mid P \mid \mu P.\, A$$

That is, separating conjunctions, disjunctions, implications, existential quantifiers, and inductive predicates are built into the framework, because they preserve backward satisfiability (as we present in Lemma 1) and we can thus reduce the effort of users of our framework to only proving backward satisfiability for input assertions; all other SL connectives (*e.g.*, magic wands) have to be provided as input assertions. Following Dardinier et al. [10], we formalize inductive predicates with one designated symbol P, which represents a recursive occurrence inside the least fixed-point computation $\mu P.\, A$. For example, the standard linked-list inductive predicate can be expressed as[8]

$$\mu P.\, \texttt{x} \neq \texttt{null} \Rightarrow (\texttt{acc(x.val)} * \texttt{acc(x.next)} * (\exists v : \texttt{Ref}.\, \texttt{v = x.next} * (\exists x : \texttt{Ref}.\, \texttt{x = v} * P)))$$

$I_\mathrm{A}^{\Delta_F}$ is extended to an interpretation function on all front-end assertions using the standard interpretation for the built-in connectives and the least fixed point interpretation for inductive predicates. fv_A, the function that returns the set of free variables for assertions, and $\mathrm{wf}_A^{\Delta_F}$, the well-formedness predicates of assertions, are extended to all front-end composite assertions accordingly.

[8]The standard linked-list predicate is usually defined as $\mathit{List(x)} \triangleq \texttt{x} \neq \texttt{null} \Rightarrow (\texttt{acc(x.val)} * \texttt{acc(x.next)} * \mathit{List}(\texttt{x.next}))$. In our syntax, the recursive occurrence P (which in our example represents *List*) does not take explicit arguments as input; we thus use an existential quantifier to bind the heap-dependent expression `x.next` to the implicit argument `x`.

The concrete definitions of the extensions of $I_A^{\Delta_F}$, fv_A , and $\mathsf{wf}_A^{\Delta_F}$ are in the extended version [32, §A].

Similarly, the parameter for statements C_p provides a set of *primitive* statements, from which the framework derives the set of statements using the following grammar, where $C_p \in C_{Fp}$, $b \in BE_F$, and A is a front-end assertion:

$$C ::= \texttt{skip} \mid C_p \mid C;C \mid \texttt{if}(b)\ \{C\}\ \texttt{else}\ \{C\} \mid \texttt{while}\ (b)\ \texttt{inv}\ A\ \{C\} \mid \{A\}\ C\ \{A\}\ ||\ \{A\}\ C\ \{A\}$$

Loops and parallel compositions contain assertions (invariants for loops, and pre- and postconditions for each parallel branch) for verification purposes. wf_C, fv_C, and wr can be extended to all front-end statements in a straightforward way, as shown in the extended version [32, §A].

The small-step operational semantic reduction rules and CSL rules for these control structures are fixed to the standard ones in the framework, as shown in the extended version [32, §A]. For primitive statements, they are expressed by the framework parameters.

To customize the state model of the input language, the framework has parameters for types (T_F), values (V_F), program heaps (H_P), and ghost heaps (H_F). Intuitively, program heaps abstract the memory part of execution states and are involved in the definition of the operational semantics of the front-end language. In contrast, ghost heaps typically include the program heaps and additional SL resources (*e.g.*, permission masks) for reasoning under SL. They are thus elements of an IDF algebra. We define front-end ghost states $\Sigma_F = (\mathsf{Var} \rightharpoonup V_F) \times H_F$ and program states $\Sigma_P = (\mathsf{Var} \rightharpoonup V_F) \times H_P$ to consist of a store of local variables and a ghost (resp. program) heap. The embedding function embed_H of heaps can then be naturally extended to an embedding $\mathsf{embed}_S : \Sigma_P \to \Sigma_F$ of program states onto ghost states.

Parameters describing the front-end translation. Our framework captures the translation from the front-end language to the IVL by parameters $[\![\cdot]\!]_\mathsf{T}$, $[\![\cdot]\!]_\mathsf{V}$, $[\![\cdot]\!]_\mathsf{H}$, $[\![\cdot]\!]_\Delta$, $[\![\cdot]\!]_\mathsf{B}$, $[\![\cdot]\!]_\mathsf{Ap}$, and $[\![\cdot]\!]_\mathsf{Cp}$ for the translation of front-end types, values, ghost heaps, type contexts, boolean expressions, input assertions, and primitive statements, respectively. The functions $[\![\cdot]\!]_\mathsf{Ap}$ and $[\![\cdot]\!]_\mathsf{Cp}$ may translate front-end input assertions and primitive statements into composite ones in the IVL.

The translation of types and values may in general lose information. For example, when translating the front-end type `Rat` of rational values into the IVL type `Real` of real values, the rationality of front-end values will not be reflected in their IVL translation. As a result, witnesses for existential quantifiers in the IVL do not necessarily witness the corresponding quantifier in the input program, which could cause the translation to be unsound.

To avoid this unsoundness, our framework offers another parameter typeInv that is used to constrain the possible witnesses in the IVL-translation. For any front-end type $t \in T_F$ and local variable $x \in \mathsf{Var}$, $\mathsf{typeInv}(t, x)$ yields a syntactic boolean IVL expression that evaluates to `true` iff the value of variable x in the IVL state has a corresponding front-end value of type t. For example when $[\![\texttt{Rat}]\!]_\mathsf{T} = \texttt{Real}$, we can define $\mathsf{typeInv}(\texttt{Rat}, x) \triangleq (\exists a : \texttt{Int}.\ \exists b : \texttt{Int}.\ b \neq 0 \wedge x = a/b)$. Using this framework parameter, we extend the translation of input assertions to all front-end assertions and obtain the assertion translation function $[\![\cdot]\!]_\mathsf{A}$. The definition of $[\![\exists x : t.\ A]\!]_\mathsf{A}$ is as follows; that of the other cases is straightforward and is listed in the extended version [32, §A].

$$[\![\exists x : t.\ A]\!]_\mathsf{A} \triangleq \exists x : [\![t]\!]_\mathsf{T}.\ \mathsf{typeInv}(t, x) * [\![A]\!]_\mathsf{A}$$

Our framework translates loops and parallel compositions in an overapproximating way as verifiers like Viper do: the loop body or each thread in the parallel composition is translated together with its annotations as a separate program; we collect these programs to form a set of *auxiliary IVL statements* $[\![c]\!]_\mathsf{C}^\mathsf{ext}$ for the front-end statement c. If c does not contain loops or parallel compositions, $[\![c]\!]_\mathsf{C}^\mathsf{ext}$ is empty. In the main program, each statement c is translated to $[\![c]\!]_\mathsf{C}^\mathsf{main}$. If c is a loop or parallel composition, $[\![c]\!]_\mathsf{C}^\mathsf{main}$ is a sequence of exhale-havoc-inhale statements which

$$[\![\texttt{while}\ (b)\ \texttt{inv}\ I\ \{c\}]\!]_C \triangleq \Big(\texttt{exhale}\ [\![I]\!]_A * ([\![b]\!]_B \vee \neg\, [\![b]\!]_B)\,;\texttt{havoc}\ \mathrm{wr}(c);\texttt{inhale}\ [\![I]\!]_A * \neg\, [\![b]\!]_B\,,$$
$$\Big\{\texttt{inhale}\ [\![I]\!]_A * [\![b]\!]_A\,;[\![c]\!]_C^{\mathrm{main}}\,;\texttt{exhale}\ [\![I]\!]_A * ([\![b]\!]_B \vee \neg\, [\![b]\!]_B)\Big\} \cup [\![c]\!]_C^{\mathrm{ext}}\Big)$$
$$[\![\{P_1\}\ c_1\ \{Q_1\}\ ||\ \{P_2\}\ c_2\ \{Q_2\}]\!]_C \triangleq \Big(\texttt{exhale}\ [\![P_1 * P_2]\!]_A\,;\texttt{havoc}\ \mathrm{wr}(c_1) \cup \mathrm{wr}(c_2);\texttt{inhale}\ [\![Q_1 * Q_2]\!]_A\,,$$
$$\Big\{\texttt{inhale}\ [\![P_1]\!]_A\,;[\![c_1]\!]_C^{\mathrm{main}}\,;\texttt{exhale}\ [\![Q_1]\!]_A\Big\} \cup$$
$$\Big\{\texttt{inhale}\ [\![P_2]\!]_A\,;[\![c_2]\!]_C^{\mathrm{main}}\,;\texttt{exhale}\ [\![Q_2]\!]_A\Big\} \cup [\![c_1]\!]_C^{\mathrm{ext}} \cup [\![c_2]\!]_C^{\mathrm{ext}}\Big)$$

Fig. 9. The translation $[\![\cdot]\!]_C$ of front-end loops and parallel compositions. The full definition of $[\![\cdot]\!]_C$ is in the extended version [32, §A]. The condition $[\![b]\!]_B \vee \neg\, [\![b]\!]_B$ in the translation of loops excludes states in which the evaluation of $[\![b]\!]_B$ fails (*e.g.*, due to a division by zero). Here we assume that the IVL supports (1) disjunction and negation of pure boolean expressions, and (2) directly using pure boolean expressions as assertions. It can therefore construct assertions including $[\![b]\!]_B \vee \neg\, [\![b]\!]_B$. **havoc** V is a shorthand for **havoc** $x_1;\cdots;$ **havoc** x_n for the finite set $V = \{x_1, \ldots, x_n\}$.

first remove the resources specified by the annotations that are required to execute the block, then assign fresh values to all local variables that the execution of the block might modify, and finally add back the resources specified by the annotations that the execution of the block yields. Given $[\![\cdot]\!]_{Cp}$ that translates front-end primitive statements into (possibly composite) IVL statements, our framework thus defines the translation $[\![c]\!]_C$ of front-end composite statement c as a pair of an IVL statement $[\![c]\!]_C^{\mathrm{main}}$ and a finite set of auxiliary IVL statements $[\![c]\!]_C^{\mathrm{ext}}$. We give the definition of $[\![\cdot]\!]_C$ for loops and parallel compositions in Fig. 9. That for the other control structures is straightforward; the full definition of $[\![\cdot]\!]_C$ is in the extended version [32, §A].

Finally, the provided parameters let us extend the encoding of front-end ghost heaps to an encoding $[\![\cdot]\!]_S$ of front-end ghost *states*: $[\![(s,\varphi)]\!]_S \triangleq ([\![s]\!]_{St}, [\![\varphi]\!]_H)$, where the store encoding $[\![s]\!]_{St}$ is defined as:

$$[\![s]\!]_{St}(x) = \begin{cases} \text{undefined}, & s(x) \text{ is undefined} \\ [\![s(x)]\!]_V, & \text{otherwise} \end{cases}$$

3.3 Requirements on the Framework Parameters

In this subsection, we present the requirements on the framework parameters that guarantee the soundness of the front-end translation. Most of these requirements for the parameters from Fig. 8 as well as the translations of types and values are standard and easy to satisfy; we list them in the extended version [32, §A].

Req. 1 states the homomorphism property of the state encoding. We state it on the ghost heap encoding parameter $[\![\cdot]\!]_H$ of the framework rather than the derived ghost state encoding $[\![\cdot]\!]_S$; the homomorphism result of $[\![\cdot]\!]_S$ can be easily derived from that of $[\![\cdot]\!]_H$ in Req. 1. The second property here is a sanity requirement that prevents the heap encoding from introducing garbage that has no correspondence in the front-end; while in the diagram of Fig. 3, backward satisfiability provides ω_A, this property guarantees the existence of ω_F and thus helps complete the desired state split in the front-end. Moreover, we require a third property, which allows us to support IDF.

Requirement 1. *The encoding of ghost heaps $[\![\cdot]\!]_H$ must satisfy the following properties:*

- *$[\![\cdot]\!]_H$ preserves addition $\oplus$ when the sum of ghost heaps is defined in the front-end,* i.e., *if $\varphi_1 \oplus \varphi_2$ is defined, then so is $[\![\varphi_1]\!]_H \oplus [\![\varphi_2]\!]_H$ and $[\![\varphi_1]\!]_H \oplus [\![\varphi_2]\!]_H = [\![\varphi_1 \oplus \varphi_2]\!]_H$.*
- *$[\![\cdot]\!]_H$ preserves subtraction in the following sense: if $[\![\varphi]\!]_H = \Phi_1 \oplus [\![\varphi_2]\!]_H$, then there exists some φ_1 such that $\Phi_1 \succeq [\![\varphi_1]\!]_H$ and $\varphi = \varphi_1 \oplus \varphi_2$, where $a \succeq b \triangleq (\exists r.\ a = b \oplus r)$.*

- $[\![\cdot]\!]_H$ *preserves stability. Specifically,* $\mathit{stabilize}([\![\varphi]\!]_H) = [\![\mathit{stabilize}(\varphi)]\!]_H$ *for any front-end ghost heap* $\varphi \in H_F$.

With the parameters of the front-end language and the translation from §3.2, we formally define backward satisfiability for front-end assertions in Def. 3.

Definition 3 (Backward satisfiability)**.** A front-end assertion A is *backward satisfiable* w.r.t. the assertion translation $[\![\cdot]\!]_{\mathsf{A}}$ iff: for any IVL state Ω that satisfies $[\![A]\!]_{\mathsf{A}}$, and that is upper-bounded by the encoding of some front-end state ω_b, *i.e.,* $[\![\omega_b]\!]_{\mathsf{S}} \succeq \Omega$, there exists a front-end state ω which satisfies A and $\Omega \oplus |[\![\omega_b]\!]_{\mathsf{S}}| \succeq [\![\omega]\!]_{\mathsf{S}}$[9].

Req. 2 formalizes that the translations of boolean expressions, assertions, and statements must preserve the semantics in the front-end. For assertions, we require in addition monotonicity of translated assertions, a property that is naturally satisfied in an affine logic, and that the translation does not add new free occurrences of P. Together with backward satisfiability, it ensures that whenever the IVL satisfies a separating conjunction, the front-end can also satisfy it by redistributing the resources between the conjuncts. We make the requirements only for input assertions in Req. 2, and prove Lemma 1 as part of the framework that these properties hold for all front-end assertions.

Requirement 2. *The translation functions for boolean expressions, input assertions, and primitive statements must satisfy the following properties:*

- *The translation* $[\![b]\!]_B$ *of any front-end boolean expression* $b \in BE_F$ *preserves the semantics of* b, i.e., *for any front-end state* $\omega \in \Sigma_F$, $I_B(b)(\omega)$ *is defined iff* $[\![b]\!]_B([\![\omega]\!]_S)$ *is defined, and they evaluate to the same truth value when both are defined.*
- *The translation of input assertions* $[\![\cdot]\!]_{Ap}$ *must satisfy:*
  - $[\![\cdot]\!]_{Ap}$ *preserves the semantics of all well-formed input assertions (*i.e., *each* A *such that* $wf_A^{\Lambda_F}(A)$*) in the front-end. Concretely, for any front-end state* ω *and any input assertion* A, ω *satisfies* A *iff* $[\![\omega]\!]_S$ *satisfies* $[\![A]\!]_{Ap}$.
  - $[\![A]\!]_{Ap}$ *is monotone for any input front-end assertion* A. *An assertion* Q *is monotone iff for any states* Ω *and* Ω', *if* Ω *satisfies* Q *and* $\Omega' \succeq \Omega$, *then* Ω' *also satisfies* Q.
  - $[\![A]\!]_{Ap}$ *of any input front-end assertion* A *does not contain any free occurrence of* P.
  - *All well-formed input assertions are backward satisfiable w.r.t.* $[\![\cdot]\!]_{Ap}$.
- *The translation of statements* $[\![\cdot]\!]_{Cp}$ *preserves the semantics in the front-end, see formalization in the extended version [32, §A].*

Lemma 1 (Extending requirements to all front-end assertions)**.** *If the translation* $[\![\cdot]\!]_A$ *is monotone, does not add new free occurrences of* P, *and preserves the semantics for front-end assertions* A_1 *and* A_2, *and these two assertions are backward satisfiable w.r.t.* $[\![\cdot]\!]_A$, *then these properties are preserved by* $[\![\cdot]\!]_A$ *of* $A_1 * A_2$, $A_1 \vee A_2$, $A_1 \Rightarrow A_2$ *(for a pure* A_1*),* $\exists x : t.\, A_1$, *and* $\mu P.\, A_1$.

As a corollary, $[\![\cdot]\!]_A$ *is monotone and preserves the semantics for all well-formed front-end assertions, and all well-formed front-end assertions are backward satisfiable w.r.t.* $[\![\cdot]\!]_A$.

In contrast, magic wands $-\!\!*$, universal quantifications $\forall$, and iterated separating conjunctions $\circledast$ do not necessarily preserve backward satisfiability of their components. For instance, assertions W, P_1, and P_2 from the two examples in §2.5 are not backward satisfiable (as explained in the corresponding examples), even though their components are by Lemma 1. For assertions containing these connectives, backward satisfiability needs to be proven without the help from Lemma 1.

[9]The stronger statement $\Omega \succeq [\![\omega]\!]_{\mathsf{S}}$ from Fig. 3 might not be satisfied by some front-ends (*e.g.*, the one encoding lazy field creation in §4.3) in the IDF algebra setting, as Ω might lose some pure information from ω_b which is necessary to construct ω.

3.4 Soundness

The conditions presented in the previous subsection are sufficient to guarantee that the front-end translation described by a given instantiation of our framework is sound. In the following, we summarize the soundness argument. The full proof is in our Isabelle/HOL formalization.

One of our central lemmas shows that the state encoding preserves the validity of separating conjunctions if the first conjunct is monotone and the second is backward satisfiable. This property is essential to show that assertions retain the meaning, no matter whether they are assumed or asserted. It is, in fact, one of the main reasons we require assertions to be backward satisfiable:

Lemma 2 (Translation of separating conjunctions)**.** *For any IVL semantic assertion A, define $btr(A) \triangleq \{\omega \mid [\![\omega]\!]_S \in A\}$ as the backward translation of A,* i.e.*, the collection of all front-end states whose encodings satisfy A.*

For arbitrary IVL semantic assertions A and B, $btr(A) * btr(B) \subseteq btr(A * B)$. Furthermore, when A is monotone, and B is the translation of some backward satisfiable front-end assertion, the other direction also holds, i.e.*, $btr(A * B) \subseteq btr(A) * btr(B)$.*

Lemma 2 allows us to prove Lemma 3 which states that the exhale-havoc-inhale translation pattern that appears in the translation of both loops and parallel compositions (as in Fig. 9) is sound. The proof proceeds by applying the CSL frame rule on the CSL proof from the first property of Lemma 3 with a carefully chosen frame assertion, followed by an application of the consequence rule where the implications at the pre- and postconditions in the consequence rule are justified by the two directions of implication in Lemma 2 respectively.

Lemma 3 (Soundness of the exhale-havoc-inhale translation pattern)**.** *Assume that the framework parameters satisfy Req. 1 and Req. 2, P_0 and Q_0 are well-formed, and that the following properties hold:*

- *$\Delta_F \vdash_{CSL} \{I_A^{\Delta_F}(P_0)\}\, c\, \{I_A^{\Delta_F}(Q_0)\}$. That is, the front-end loop or parallel composition block has a proof under CSL against the pre- and postconditions from the annotation of the block.*
- *$[\![\Delta_F]\!]_\Delta \vdash [P]$ `exhale` $[\![P_0]\!]_A$; `havoc` $wr(c)$; `inhale` $[\![Q_0]\!]_A$ $[Q]$. That is, the main program of the translation of the loop or parallel composition verifies against another given pair of pre- and postconditions in the IVL.*
- *The given IVL pre- and postconditions P and Q are monotone.*
- *All written variables of the loop or parallel composition block are declared, and thus included in the type context,* i.e.*, $wr(c) \subseteq \mathrm{dom}(\Delta_F)$.*

Then $\Delta_F \vdash_{CSL} \{btr(P)\}\, c\, \{btr(Q)\}$. That is, the front-end CSL proof for the block can be lifted to against the new pre- and postconditions backward translated from the IVL.

We can then prove our central soundness result stated as Thm. 4. The proof proceeds by induction on the structure of front-end statements, where for the loop and parallel composition cases, we can apply Lemma 3 with the IVL assertions P and Q in the lemma as the translation of the desired front-end pre- and postconditions $[\![P]\!]_A$ and $[\![Q]\!]_A$ and the first property derived by applying induction hypotheses on the axiomatic semantic proofs of the auxiliary IVL statements and CSL loop rule or parallel rule subsequently.

Theorem 4 (Soundness)**.** *Assume that the framework parameters satisfy Req. 1 and Req. 2. Let c be a well-formed front-end composite statement w.r.t. type context Δ_F, and $[\![c]\!]_C = (C_m, C_e)$ be the translation of c. Let P and Q be arbitrary well-formed front-end composite assertions. Assume (1) $[\![\Delta_F]\!]_\Delta \vdash [[\![P]\!]_A]\, C_m\, [[\![Q]\!]_A]$, and (2) any $C \in C_e$ verifies in the IVL,* i.e.*, there exists an IVL semantic assertion Q_C such that $[\![\Delta_F]\!]_\Delta \vdash [\top]\, C\, [Q_C]$. Then $\Delta_F \vdash_{CSL} \{I_A^{\Delta_F}(P)\}\, c\, \{I_A^{\Delta_F}(Q)\}$ holds in CSL.*

This theorem guarantees that, whenever the IVL translation of a front-end program is correct then there exists a proof in CSL for the front-end program. To further connect this result to the

operational semantics of the front-end language, we also prove the soundness of the CSL rules w.r.t. the small-step semantics. Our proof adapts the proof from Vafeiadis [56] to our IDF setting. The formal statement of this result is in the extended version [32, §A].

3.5 Generalization to State Encoding Relations

As we have seen in §2.1, some front-ends might relate a single front-end state with multiple IVL states, as in the case of encoding duplicable permissions. To capture such encodings, we generalize the state encoding from a function $[\![\cdot]\!]_{\mathsf{S}}$ to a binary relation $\cdot \sim_{\mathsf{S}} \cdot$ between front-end states and IVL states and write $\omega \sim_{\mathsf{S}} \Omega$ if IVL state Ω is *one of the possible encodings* of front-end state ω.

The generalized framework takes the same inputs as in Fig. 8, except that the heap encoding function $[\![\cdot]\!]_{\mathsf{H}}$ is replaced by a binary relation $\cdot \sim_{\mathsf{H}} \cdot$ that models the encoding of front-end ghost heaps into IVL heaps. The framework extends $\cdot \sim_{\mathsf{H}} \cdot$ naturally to the binary relation $\cdot \sim_{\mathsf{S}} \cdot$ between front-end and IVL *states*: $(s, h) \sim_{\mathsf{S}} (S, H) \triangleq (S = [\![s]\!]_{\mathsf{St}} \wedge h \sim_{\mathsf{H}} H)$.

Most requirements of our framework are not affected by this generalization, but Req. 1 and Req. 2 need to be adapted:

Requirement 1′. *The binary relation $\cdot \sim_H \cdot$ between front-end ghost heaps and IVL heaps need to satisfy the following properties.*

- *If $\varphi = \varphi_1 \oplus \varphi_2$ and $\varphi \sim_H \Phi$, then there exist Φ_1 and Φ_2 such that $\Phi = \Phi_1 \oplus \Phi_2$, $\varphi_1 \sim_H \Phi_1$, and $\varphi_2 \sim_H \Phi_2$.*
- *If $\varphi \sim_H \Phi$ and $\varphi_0 = \varphi \oplus \varphi_1$, then there exists Φ_1 such that $\varphi_1 \sim_H \Phi_1$, $\Phi \oplus \Phi_1$ is defined, and $\varphi_0 \sim_H (\Phi \oplus \Phi_1)$.*
- *For every front-end ghost heap $\varphi \in H_F$, there exists an IVL heap $\Phi \in \Sigma$ such that $\varphi \sim_H \Phi$.*
- *If $\varphi \sim_H \Phi$, then stabilize$(\varphi) \sim_H$ stabilize(Φ).*
- *If stabilize$(\varphi) \sim_H \Phi'$, then $\exists \Phi$ such that stabilize(Φ) = stabilize(Φ') and $\varphi \sim_H \Phi$.*

Compared with Req. 1 in §3.3, the adaptation is in the following ways:

- The first property in Req. 1 (preservation of addition) is adjusted in a straightforward way.
- The second property in Req. 1 (preservation of subtraction) is now subsumed by a generalized backward satisfiability, see below.
- The second property here excludes certain ill-formed encodings by requiring that whenever a heap can be extended in the front-end, the corresponding extension is possible in the IVL (for instance, it is not possible to encode a fractional permission in the front-end to a full permission in the IVL). For the case where the heap encoding is a function, this property follows from the addition requirement and is, thus, not explicitly required in Req. 1.
- The third property here, intuitively, ensures that the encoding is non-degenerated.
- The preservation of stability (*i.e.*, the last requirement in Req. 1) is adapted to the last two requirements here.

We adjust the definition of backward satisfiability to Def. 5. The definition coincides with the intuition in Fig. 3, except that now ω_F may be encoded into an even larger Ω'_F rather than Ω_F. This relaxation gives more flexibility with no affection on the soundness of the framework, intuitively because excess resources can always be discarded harmlessly in an affine SL.

Definition 5 (Generalized backward satisfiability)**.** A front-end assertion A has *generalized backward satisfiability* w.r.t. the assertion translation $[\![\cdot]\!]_{\mathsf{A}}$ iff: for any front-end state ω and IVL states Ω, Ω_A, and Ω_F, if $\omega \sim_{\mathsf{S}} \Omega$, $\Omega = \Omega_A \oplus \Omega_F$, and Ω_A satisfies $[\![A]\!]_{\mathsf{A}}$, then there exist front-end states ω_A and ω_F and IVL state Ω'_F such that: $\omega = \omega_A \oplus \omega_F$, ω_A satisfies A, $\omega_F \sim_{\mathsf{S}} \Omega'_F$, and $\Omega'_F \succeq \Omega_F$.

With the generalized backward satisfiability notion, Req. 2 from §3.3 needs only technical adaptations:

Requirement 2′. *The translation functions for boolean expressions, input assertions, and primitive statements must satisfy the following properties:*

- *The translation* $[\![b]\!]_B$ *of any front-end boolean expression* $b \in BE_F$ *preserves the semantics of* b, i.e., *for any front-end state* $\omega \in \Sigma_F$ *and IVL state* $\Omega \in \Sigma_V$ *with* $\omega \sim_S \Omega$, $I_B(b)(\omega)$ *is defined iff* $[\![b]\!]_B(\Omega)$ *is defined, and they evaluate to the same value when both are defined.*
- *The translation of input assertions* $[\![\cdot]\!]_{Ap}$ *must satisfy:*
  - $[\![\cdot]\!]_{Ap}$ *preserves the semantics of all well-formed input assertions in the front-end. Concretely, for any front-end state* ω *and IVL state* Ω *with* $\omega \sim_S \Omega$, ω *satisfies* A *iff* Ω *satisfies* $[\![A]\!]_{Ap}$.
  - $[\![A]\!]_{Ap}$ *is monotone for any* $A \in A_{Fp}$.
  - $[\![A]\!]_{Ap}$ *of any input front-end assertion* A *does not contain any free occurrence of* P.
  - *Any well-formed* $A \in A_{Fp}$ *has generalized backward satisfiability w.r.t.* $[\![\cdot]\!]_{Ap}$.
- *The translation of statements* $[\![\cdot]\!]_{Cp}$ *preserves the semantics in the front-end.*

To show that our framework for state encoding relations is indeed a generalization, we instantiate it, modeling the state encoding as a function, as stated in Thm. 6.

Theorem 6 (Generalization)**.** *Given an instantiation of the generalized framework where the state encoding is a function, if this instantiation satisfies all requirements from §3.3 then it also satisfies the adapted requirements of the generalized framework.*

Our main result is that any instantiation of the generalized framework that satisfies the adapted requirements is *sound*. That is, Thm. 4, with the adapted requirements, also holds in the generalized framework.

The proof of the theorem requires several adaptations. In the generalized setting, we define the backward translation of IVL semantic assertion P as $\mathrm{btr}(P) \triangleq \{\omega \mid \exists \Omega \in P.\, \omega \sim_S \Omega\}$, and call P well-formed iff $\mathrm{btr}(P) = \{\omega \mid \forall \Omega.\, \omega \sim_S \Omega \Rightarrow \Omega \in P\}$. The direction $\mathrm{btr}(A) * \mathrm{btr}(B) \subseteq \mathrm{btr}(A * B)$ in Lemma 2 does not hold for general A and B anymore, but requires B to be well-formed. As a result, Q_0 being well-formed is needed for Lemma 3. Despite these changes, we prove that all the assertions supported by a front-end instantiation of our generalized framework are well-formed as long as the instantiation satisfies all the requirements in §3.5. As a result, the new premise in Lemma 3 is always satisfied, and we get the same soundness result as Thm. 4 in the generalized framework.

The generalized framework provides an expressive foundation for proving the soundness of practical translational verifiers. In particular, it reduces the soundness proof for state encodings to proving a small number of relatively simple requirements. We will demonstrate this expressiveness in §4.4 by instantiating it with an encoding of duplicable permissions for immutable location.

4 Applications of the Framework

In this section, we demonstrate the applicability of our framework by formalizing three front-end translations with different state encodings into Viper, whose state model we formally define in §4.1. We first revisit the encoding from rational permission amount to reals (§4.2), and we show how to extend this encoding to support a backward-satisfiable fragment of magic wands used by Prusti [1]. We also translate a language with lazy field creation (§4.3), and show an encoding of an immutable heap (§4.4). We describe more encodings inspired by real-world front-ends (§4.5). Finally, we discuss previously-reported soundness bugs in existing translational verifiers whose root causes manifest themselves as violations of our framework's requirements (§4.6).

In all the examples, our framework substantially simplifies the effort required for defining the translation and proving it sound: a formalization without our framework would essentially have to repeat the definitions and proofs of our main theorems, in addition to proving the requirements

of our framework. Our framework also makes the soundness proof easy to maintain in practice, as it decomposes the *global* soundness of a front-end translation into *local* proof obligations: the translation of each input assertion must be backward satisfiable, and the translation of each primitive statement must be sound. This modularity keeps the proof robust under changes: adding a new input assertion (*e.g.*, the sound magic wand fragment in §4.2) does not require revisiting the overall soundness argument, only proving the corresponding requirements for that assertion.

4.1 Viper State Model

The formalization of Viper we use models the heap as a partial function from heap locations to values and the currently-owned permissions as a partial function from heap locations to real numbers, called *permission mask*. A heap h with a permission mask π forms an IDF algebra with the following instantiations: $\Sigma = \{(h, \pi) \in (L \rightharpoonup V) \times (L \to [0,1]) \mid \forall l.\, \pi(l) > 0 \Rightarrow h(l) \neq \bot\}$, where a heap location $l \in L$ is a pair of an address (modeled as natural number) and a field (identified by a unique name); $\oplus$ adds up the permission amounts at each heap location and combines the values in the heap when the two heaps are compatible (the sum of permissions at each heap location does not exceed 1, and the two heaps do not have different values at any heap location), *i.e.*, $(h_1, \pi_1) \oplus (h_2, \pi_2)$ is defined iff $\forall l.\, \pi_1(l) + \pi_2(l) \leq 1 \wedge (l \in \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) \Rightarrow h_1(l) = h_2(l))$, and equals to $(h_1 \cup h_2, \pi_1 + \pi_2)$ when it is defined; $|\_|$ clears all the permissions, *i.e.*, $|(h, \pi)| \triangleq (h, \lambda l.\, 0)$; stabilize erases all values for heap locations to which no permission is held, *i.e.*, $\mathsf{stabilize}((h, \pi)) \triangleq (\lambda l.\, \text{if } \pi(l) > 0 \text{ then } h(l) \text{ else } \bot, \pi)$; stable asserts whether every heap location storing a concrete value has a positive permission amount, *i.e.*, $\mathsf{stable}((h, \pi)) \Leftrightarrow (\forall l.\, h(l) \neq \bot \Rightarrow \pi(l) > 0)$; unit erases the whole heap with the permissions, *i.e.*, $\mathsf{unit}((h, \pi)) = (\lambda l.\, \bot, \lambda l.\, 0)$;

A Viper state is thus an IVL state where the customized heap is instantiated with a heap with a permission mask, *i.e.*, $\Sigma_V = (\mathsf{Var} \rightharpoonup V) \times \Sigma$ with Σ defined above.

4.2 Encoding Rational Permission Amounts into Real Permission Amounts

In §2.5, we showed that the common encoding of fractional permissions with rational permission amounts into real permission amounts is generally unsound. Our example there made use of magic wands. In this subsection, we show that such an encoding is sound for a front-end with a more restricted assertion language.

To focus on the essentials, we assume that the front-end language is very similar to our IVL, Viper. Both have the same types (*e.g.*, for booleans, integers, and references), values (*e.g.*, `null`), and expressions (except that the front-end uses rational arithmetic whereas the IVL uses reals). Among the primitive statements, local variable assignment and field assignment are common to front-end language and IVL; moreover, the front-end supports allocation and de-allocation, whose translation is discussed below.

We provide the details of the instantiation of our framework in the extended version [32, §B.1] and focus on the most interesting aspects here. The front-end models the program heap as a partial map from heap locations (as pairs of natural number identifiers and string-named fields, which is the same as in Viper) to values, and uses rational permission amounts for its ghost heap, *i.e.*, $H_P = \{h_p : L \rightharpoonup_{\mathsf{fin}} V_F\}$ where $\rightharpoonup_{\mathsf{fin}}$ represents a finite partial mapping , and $H_F = \{(h, \pi) \in (L \rightharpoonup V_F) \times (L \to [0,1] \cap \mathbb{Q}) \mid \forall l.\, \pi(l) > 0 \Rightarrow h(l) \neq \bot\}$. The embedding of H_P onto H_F keeps the partial heap, and assigns full permissions to heap locations that contain values and zero permissions otherwise: $\mathsf{embed}_H(h_p) = (h_p, \lambda l.\ \text{ if } h_p(l) \neq \bot \text{ then } 1 \text{ else } 0)$.

Due to the similarity of the front-end language and IVL, the translation functions for types, values, ghost heaps, and expressions are straightforward, except that the type `Rat` is translated to `Real`. $\mathsf{typeInv}(t, x)$ is thus non-trivial only for $t =$ `Rat` as $\mathsf{typeInv}(\texttt{Rat}, x) \triangleq \exists a : \texttt{Int}.\, \exists b : \texttt{Int}.\, b \neq 0 \wedge x = a/b$.

The input assertion $\mathbf{acc}(x.f, p)$ is translated unchanged to $\mathbf{acc}(x.f, p)$. Preservation of semantics of the translation and monotonicity of the translated assertion are straightforward. Backward satisfiability is also intuitive: if $[\![p]\!]_{\mathsf{E}}$[10] evaluates to v under some Ω with $[\![\omega_b]\!]_{\mathsf{S}} \succeq \Omega$, then it also evaluates to v under $[\![\omega_b]\!]_{\mathsf{S}}$ (adding resources cannot change a defined evaluation), and hence to v under ω_b in the front-end, so v is rational. The front-end state holding just v permission to $x.f$ (and only the information needed to evaluate x and p) is therefore the smallest state satisfying $\mathbf{acc}(x.f, p)$, and its encoding is smaller than or equal to Ω.

The translations of variable assignment and field assignment are straightforward. Allocation and de-allocation are translated as follows:

$$[\![x := \mathbf{alloc}(f_1, \ldots, f_n)]\!]_{\mathsf{Cp}} \triangleq \mathbf{havoc}\ x; \mathbf{inhale}\ \mathbf{acc}(x.f_1, 1) * \cdots * \mathbf{acc}(x.f_n, 1)$$
$$[\![\mathbf{free}(x.f)]\!]_{\mathsf{Cp}} \triangleq \mathbf{exhale}\ \mathbf{acc}(x.f, 1)$$

Intuitively, the translation of allocation picks non-deterministically a fresh heap location identifier ($\mathbf{havoc}\ x$) and provides the necessary permissions for the allocated heap locations; the translation of de-allocation removes the permission to the deleted heap location.

We prove in Isabelle/HOL that this instantiation satisfies all requirements of our framework. This instantiation shows that translating rational permission amounts as reals is not per se unsound. If we had added general magic wands as input assertions to our instantiation (as in Fig. 7 in §2.5) then the proof would have failed because Req. 2 would not hold. Nevertheless, our framework can still support a restricted fragment of magic wands.

A sound fragment of magic wands. In practice, the magic wands used by existing verifiers are often far more limited. Prusti [1], for instance, uses only wands whose two sides are *binary assertions* (*i.e.*, never use fractional permissions), and that do not access rational-valued fields. Such assertions are given by the following grammar (we further check that the fields they read do not have the type $\mathbf{Rat}$ with the framework's well-formedness predicate $\mathsf{wf}_A^{\Delta_F}$), where $e \in BE_F$ and $t \in T_F$:

$$A_b ::= \mathbf{acc}(x.f, 1) \mid e \mid A_b * A_b \mid A_b \vee A_b \mid e \Rightarrow A_b \mid \exists x : t.\, A_b$$

Each such assertion is translated like the corresponding case of $[\![\cdot]\!]_{\mathsf{A}}$ in the extended version [32, §A]; we write $[\![\cdot]\!]_{\mathsf{Ab}}$ for this translation. We then add to the front-end's input assertions all wands $L \mathrel{-\!\!*} R$ whose two sides are binary assertions that do not access rational fields, with the standard interpretation and $[\![L \mathrel{-\!\!*} R]\!]_{\mathsf{Ap}} \triangleq [\![L]\!]_{\mathsf{Ab}} \mathrel{-\!\!*} [\![R]\!]_{\mathsf{Ab}}$. We prove in Isabelle/HOL that $[\![L \mathrel{-\!\!*} R]\!]_{\mathsf{Ap}}$ satisfies Req. 2, so this fragment of magic wands is captured by our framework and, thus, sound.

4.3 Encoding Lazy Field Creation

In §1, we showed an encoding of a Python-like front-end language with lazy field creation into an IVL with static field declarations. In this subsection, we define a front-end that formalizes the lazy field creation feature, and a translation into Viper. Our framework is able to capture this translation to prove its soundness. This instantiation shows, in particular, that our framework is able to express translations with different heap structures (each front-end field is encoded as two IVL fields) and custom proof rules (as we explain below).

We assume a front-end similar to the one from §4.2. Its logic reasons about lazy field creation as follows: It governs access to a location $x.f$ using the standard accessibility predicate $\mathbf{acc}(x.f, p)$. We assume that the front-end can determine statically which fields are *potentially* added to an object, for instance, by using a type system such as Python's MyPy. Allocation provides permission to all those fields and sets their value to $\mathbf{nofield}$ to indicate that this field has not been created yet. Field updates require permission to access the field, as usual. The first assignment to a field

[10]We use $[\![\cdot]\!]_{\mathsf{E}}$ to denote the translation of general expressions (including non-boolean ones). It is not a parameter of our framework, but framework parameters can use it (like any other mathematical function).

creates the field, which is reflected by replacing **nofield** with the assigned value. Finally, field read requires permission and asserts that the field value is different from **nofield**, as attempting to read a field before it has been added triggers a runtime error. Since **nofield** is not part of the surface syntax, the frontend provides pure input assertion **added**$(x.f)$ and **missing**$(x.f)$ to denote $x.f \neq$ **nofield** and $x.f =$ **nofield**, respectively.

To reflect this logic, the front-end provides the designated value **nofield** and uses it in its heap model: $H_P = \{h_p : L \rightharpoonup_{\text{fin}} V_F \cup \{\textbf{nofield}\}\}$ and $H_F = \{(h, \pi) \in (L \rightharpoonup V_F \cup \{\textbf{nofield}\}) \times (L \to [0,1] \cap \mathbb{Q}) \mid \forall l.\, \pi(l) > 0 \Rightarrow h(l) \neq \bot\}$. The evaluation of a heap access expression $x.f$ is defined only if its value is different from **nofield**. Other than that, the evaluation of expressions is the same as in §4.2.

Our front-end supports the input assertions **acc**$(x.f, p)$, **added**$(x.f)$, and **missing**$(x.f)$. Compared with the interpretation of **acc**$(x.f, p)$ in the previous subsection, the interpretation here additionally accepts the case where the field is missing, but excludes the case where p evaluates to 0. The interpretation of **added**$(x.f)$ and **missing**$(x.f)$ is straightforward. The concrete definition of $I_{\mathsf{A}}^{\Delta_F}$ for the three input assertions is shown in the extended version [32, §B.2].

Local variable assignments, allocations, and de-allocations have the same form as in the previous subsection. For simplicity, we restrict field assignments to the form $x.f := v$, where $v \in \text{Var}$ is a local variable. The CSL rules for the primitive statements are the same as before, and so are the operational semantic reduction rules for local variable and field assignments and de-allocations. For an allocation $x :=$ **alloc**$(f_1, \ldots, f_n)$, the allocated locations are initialized to **nofield** rather than arbitrary concrete values. The formal reduction rule is shown in the extended version [32, §B.2].

The most interesting aspect of this front-end is its state encoding. Each front-end field f is encoded by *two* IVL fields: The boolean field f_{added} indicates whether the field f has been added, whereas the field f_{IVL} stores the translated value of the front-end heap location. That is, our encoding reflects the cases where a field has been added or not as follows (v is different from **nofield**).

$$a.f \overset{p}{\mapsto} \textbf{nofield} \text{ for } p \in [0,1] \qquad a.f_{added} \overset{p}{\mapsto} \textbf{false},\ a.f_{IVL} \overset{0}{\mapsto} \bot$$

$$a.f \overset{p}{\mapsto} v \text{ for } p \in [0,1] \qquad a.f_{added} \overset{p}{\mapsto} \textbf{true},\ a.f_{IVL} \overset{p}{\mapsto} [\![v]\!]_{\mathsf{V}}$$

Expressions are translated identically, except that every field name f in field access expressions is replaced by f_{IVL}. The translation of input assertions needs to reflect the above state encoding. In particular, $[\![\textbf{acc}(x.f, p)]\!]_{\mathsf{A}}$ denotes permission to f_{added} and, in case the field has been added, also permission to f_{IVL}:

$$[\![\textbf{acc}(x.f, p)]\!]_{\mathsf{Ap}} \triangleq [\![p]\!]_{\mathsf{E}} > 0 * \textbf{acc}([\![x]\!]_{\mathsf{E}}.f_{added}, [\![p]\!]_{\mathsf{E}}) * ([\![x]\!]_{\mathsf{E}}.f_{added} = \textbf{true} \Rightarrow \textbf{acc}([\![x]\!]_{\mathsf{E}}.f_{IVL}, [\![p]\!]_{\mathsf{E}}))$$

Field assignments are translated as follows:

$$[\![x.f := v]\!]_{\mathsf{Cp}} \triangleq \textbf{if}([\![x]\!]_{\mathsf{E}}.f_{added} = \textbf{false})\ \{\textbf{inhale acc}([\![x]\!]_{\mathsf{E}}.f_{IVL}, 1)\}\ \textbf{else}\ \{\textbf{skip}\};$$
$$[\![x]\!]_{\mathsf{E}}.f_{added} := \textbf{true}; [\![x]\!]_{\mathsf{E}}.f_{IVL} := v$$

Intuitively, it uses the field $[\![x]\!]_{\mathsf{E}}.f_{added}$ to check whether this assignment dynamically creates the field. If so, it adds full permission to the field $[\![x]\!]_{\mathsf{E}}.f_{IVL}$, which enables future read and write accesses. Moreover, it records in $[\![x]\!]_{\mathsf{E}}.f_{added}$ that the field has been created (in either case, as this is a monotonic update), and assigns the value to $[\![x]\!]_{\mathsf{E}}.f_{IVL}$. The translation of the other primitive statements is straightforward; the full definition is presented in the extended version [32, §B.2].

The front-end translation described above satisfies all requirements of our framework (for state encoding functions) and is, thus, guaranteed to be sound.

4.4 Encoding Immutable Heaps

Our final instantiation illustrates how our framework supports duplicable resources into an IVL that offers only (non-duplicable) fractional resources. Duplicable resources have many applications, for

instance, as permission to acquire a lock or to represent monotonic information. In this subsection, we use access to immutable data as an example for duplicable resources.

To avoid the complications of initializing immutable fields, we assume a front-end here, where immutable data is stored in a separate heap. Field accesses select the heap using different dereferencing operators, $x.f$ for the mutable heap and $x_\circ f$ for the immutable heap.[11] Allocation in the mutable heap works as usual, whereas allocation in the immutable heap takes as arguments the initial values of the fields and initializes them. Hence, regular allocation provides the usual ownership of the allocated locations, donated again by $\mathbf{acc}(x.f, p)$, whereas immutable allocation provides a *duplicable* resource $\mathbf{init}(x_\circ f)$, which permits read access, but not assignments.

To reflect these ideas, our front-end is similar to the one in §4.2, but has two heaps: $H_P = \{(h, h_{im}) \in (L \rightharpoonup V_F) \times (L \rightharpoonup V_F)\}$, where the first component is the mutable partial heap and the second one is the immutable heap. The ghost heap set is $H_F = \{(h, \pi, h_{im}) \in (L \rightharpoonup V_F) \times (L \to [0,1] \cap \mathbb{Q}) \times (L \rightharpoonup V_F) \mid \forall l.\, \pi(l) > 0 \Rightarrow h(l) \neq \bot\}$. The embedding function embed_H maps the immutable heap h_{im} identically and the mutable heap h as to a heap (h, π) with rational permissions. The resource of each heap location in the immutable heap is stable and duplicable, *i.e.*, the immutable heap in H_F is instantiated as an IDF algebra with the following definitions of the operations:

- $h_1 \oplus h_2$ is defined iff $\forall l \in L$, if both $h_1(l)$ and $h_2(l)$ are defined then $h_1(l) = h_2(l)$. In this case, $(h_1 \oplus h_2)(l) \triangleq$ if $h_1(l)$ is defined then $h_1(l)$ else $h_2(l)$.
- $|h| = \mathsf{stabilize}(h) \triangleq h$, and $\mathsf{stable}(h)$ holds for any h.
- $\mathsf{unit}(h) \triangleq \mathit{emp}$, where emp is the empty heap $\lambda l.\, \bot$.

The front-end supports the same primitive statements as in §4.2. In addition, the statement $x := \mathbf{immalloc}(f := v)$ allocates and initializes an immutable location $x_\circ f$. The operational semantic reduction rule and CSL rule for this statement are straightforward; we list them in the extended version [32, §B.3].

Since our IVL does not provide duplicable resources, we encode them as arbitrary positive fractional permission amounts. As a result, the representation for a duplicable front-end resource is not unique. Our generalized framework allows us to express such state encoding relations.

The encoding of front-end ghost heaps is modeled as a binary relation. Both front-end heaps are encoded into a single IVL heap. To disambiguate accesses, we use a map f_h that maps each front-end field f in the mutable heap to the IVL field hf; the map f_v analogously maps each front-end field g in the immutable heap to the IVL field vg. The ranges of f_h and f_v are disjoint. The encoding of the mutable heap is the same as in §4.2 (except that here field names are mapped). The (allocated) heap locations in the immutable heap are encoded to having *any* permission $p \in (0, 1)$; that is, the state relation permits *all* such amounts p. Un-allocated locations have zero permission and no value.

To formalize the state encoding relation, we need to divide the IVL heap into the parts corresponding to the mutable and immutable front-end heaps. In those parts, we then undo the field name mappings f_h and f_v so that we can relate them to the front-end fields. Concretely, we define $H|_{f_h}$ to be the IVL sub-heap constructed by taking only locations with field names in the range of f_h; we construct $H|_{f_v}$ analogously. H_{rem} denotes the remaining part of H after removing $H|_{f_h}$ and $H|_{f_v}$. $\Pi|_{f_h}$, $\Pi|_{f_v}$, and Π_{rem} are defined similarly for the permission mask Π (while "removing" means setting the permission to each location in the corresponding part to zero). Now we can define the state relation, by reusing the ghost heap encoding $[\![\cdot]\!]_{\mathsf{H}}$ from §4.2 for the mutable heap:

$$\begin{aligned}(h, \pi, h_{im}) \sim_{\mathsf{H}} (H, \Pi) \triangleq\ & \left(H|_{f_h}, \Pi|_{f_h}\right) = [\![(h, \pi)]\!]_{\mathsf{H}} \wedge H_{\mathsf{rem}} = \lambda l.\, \bot \wedge \Pi_{\mathsf{rem}} = \lambda l.\, 0 \wedge (\forall l.\\ & (h_{im}(l) = \bot \Rightarrow (H|_{f_v}(l) = \bot \wedge \Pi|_{f_v}(l) = 0)) \wedge\\ & (h_{im}(l) = v \Rightarrow (H|_{f_v}(l) = [\![v]\!]_{\mathsf{V}} \wedge \Pi|_{f_v}(l) \in (0, 1))))\end{aligned}$$

[11] Note that both heaps share the same identifiers, so a field f may occur in both heaps, and a syntactic distinction of the two accesses is necessary.

Heap access expressions are translated accordingly: the field names for all accesses to the mutable heap (of the form $x.f$) are translated by f_h, while those for the accesses to the immutable heap (of the form $x_\circ f$) are translated by f_v. The translation for the input assertions are as follows, where $\texttt{acc}(x.f, \_)$ represents ownership of an arbitrary positive fractional permission amount for $x.f$.

$$[\![\texttt{acc}(x.f, p)]\!]_{\text{Ap}} \triangleq \texttt{acc}([\![x]\!]_{\text{E}}.f_h(f), [\![p]\!]_{\text{E}}) \qquad [\![\texttt{init}(x_\circ f)]\!]_{\text{Ap}} \triangleq \texttt{acc}([\![x]\!]_{\text{E}}.f_v(f), \_)$$

The translation of local variable assignments, allocations in the mutable heap, and de-allocations is similar to that in §4.2; we show them in the extended version [32, §B.3]. The translation of allocation in the immutable heap assigns a fresh heap location, adds the resource corresponding to $\texttt{init}(x_\circ f)$ to the state, and assumes that the new location has its initial value:

$$[\![x := \texttt{immalloc}(f := v)]\!]_{\text{Cp}} \triangleq \texttt{havoc}\ x; \texttt{inhale}\ \texttt{acc}(x.f_v(f), \_) * x.f_v(f) = v$$

We proved in Isabelle/HOL that the instantiation of our *generalized* framework for immutable data satisfies all requirements and is, thus, guaranteed to be sound. This instantiation also demonstrates that our generalized framework is sufficiently expressive to handle complex, but common state encodings such as translating duplicable resources to non-duplicable fractional resources.

4.5 Additional Encodings Inspired by Real-World Front-Ends

Beyond the three instantiations formalized above, we believe that our framework can capture many other front-ends. To illustrate this point, we outline three further state encodings, inspired by real-world front-ends, that also fit our framework:

The Pancake verifier's state model [61]: The front-end heap is a fixed-size array of cells, encoded into a Viper sequence `heap` of references, where the cell at offset `i` corresponds to the heap location `heap[i]` refers to. It uses the same fractional permission model as Viper.

The block-offset memory model for C, adapted from Leroy [30]: C memory is modeled as an unbounded but finite set of blocks, each with a unique identifier, a linear array of cells (with binary permissions) indexed by their offset, and a fixed size. The encoding maps each cell (identified by its block and offset) to a unique heap location storing its value in a `val` field, and each block to a heap location whose `size` field stores the block's size.

The object-oriented memory model from Lööw et al. [33]: The front-end heap consists of an unbounded but finite set of objects, each with a unique identifier, a list of cells (with binary permissions) named by field, and a domain (the set of its field names). The encoding maps each object to a unique heap location, each cell to the field of that location with the same name, and stores the object's domain in an additional field.

In all three instantiations, the homomorphism requirement holds: Two heaps in the Pancake state model must agree on the array size and the content of each cell, allowing the combination of the encodings; the permissions are simply added for each cell. In the other two models, two states with the same identifier must agree on their shared contents and have disjoint domains on the non-duplicable contents, so their encodings combine: two C blocks must agree on their size and two objects on their domain; blocks or objects from two states must own disjoint cells. Backward satisfiability of the input assertions (*e.g.*, fractional ownership of a cell, or a block's size) is likewise straightforward.

4.6 Identifying Unsoundness in Real-World Front-End Translations

In addition to proving existing front-ends sound, our framework captures common sources of unsoundness in existing verifiers: the root causes of several previously-reported soundness bugs correspond to violations of our requirements, showing that checking these requirements could have caught them. For example, issues #1189 [48] and #1518 [49] in Prusti [1] are caused by passing the same shared reference as two arguments to a method call, which creates two independent

fractional permissions in the IVL, whereas the front-end state model forbids such a split, which thus violates the homomorphism requirement in Req. 1. Violations of Req. 2 underlie issue #955 [54] (an unsound quantifier-simplification rewrite) in VerCors [4], issue #1501 [50] (wrong encoding of `matches!` inside assertions) in Prusti, and issue #281 [51] (unsound bool/object round-trip cast) in Nagini [14]. Violations of requirements in the extended version [32, §A] underlie further bugs: soundness of the front-end SL rules is violated by issue #703 [53] (an unreachable send is treated as transferring permission) in VerCors, injectivity of value translation by issue #1357 [55] (`int` and `short` are both mapped to the same IVL type) in VerCors, and soundness of the statement translation by issue #161 [52] (static field updates are not propagated) in Nagini.

5 Related Work

Our work was inspired by Dardinier et al. [11]'s framework for proving the soundness of SL-based translational verifiers. However, their framework requires the front-end and the IVL to have the same state model, which is not the case in existing translational SL-verifiers, such as Gobra, Nagini [14], a Pancake verifier [61], Prusti, and VeriFast. In contrast, our framework supports complex state encodings, both in terms of the representation of values and of SL resources.

Several existing works target the soundness of front-end translations. For instance, Summers and Müller [47] and Wolf et al. [60] sketch soundness proofs for the Viper translations of the RSL weak memory logic [57] and the TaDA logic [9], respectively. Both involve non-standard SL resources, for which our framework provides a formal foundation.

Instead of translating state models to the IVL, Gillian uses an IVL and verification back-end that are parametric in the state model. This simplifies front-end soundness proofs but requires the back-end to be sound for all instantiations. Lööw et al. [33] present a framework for such proofs in Rocq, which is parametric in the concrete and symbolic state models, with sufficient conditions for a sound instantiation. However, unlike our work, it includes no statement or assertion translations and supports neither loops nor concurrency, so it is unclear whether it can capture realistic front-end translations. Moreover, it does not facilitate soundness proofs for IVLs that are not parametric in the state model, such as Raven, VeriFast, and Viper.

Front-end translations to IVLs not based on SL [58, 3, 20, 16, 38] involves generally much simpler state encodings. In fact, the existing unsound encodings from §2.2 and §2.5 as well as our sufficient soundness conditions are all concerned with the representation of SL resources.

Some existing approaches target the soundness of SL-based verification back-ends. Parthasarathy et al. [38] present a technique for automatically producing certificates for successful verification runs in Viper. Zimmerman et al. [62] present a formalization of a variant of Viper's symbolic execution back-end for gradual verification. Jacobs et al. [23] formalize and prove sound a simplified version of the symbolic execution back-end of VeriFast [22] in Rocq. These back-end soundness proofs can be combined with our framework's front-end proofs for an overall soundness guarantee.

6 Conclusion and Future Work

We presented the first framework, formalized in Isabelle/HOL, for proving the soundness of the front-ends of translational separation logic verifiers with complex state encodings, which are pervasive in existing verifiers. Our sufficient soundness conditions greatly simplify soundness proofs and help uncover flawed encodings quickly, as we illustrate on several common encodings.

As future work, we plan to extend the framework to support additional concurrency primitives (*e.g.*, threads and locks) and to apply the framework to prove soundness of existing front-ends.

Data Availability Statement

All technical results presented in this paper have been formalized and proved in Isabelle/HOL. The formalization is publicly available in our artifact [31]. The latest version is available at https://doi.org/10.5281/zenodo.21509501.

References

[1] 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 (Oct. 2019), 147:1–147:30. doi:10.1145/3360573

[2] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 2025. A Hybrid Approach to Semi-Automated Rust Verification. arXiv:2403.15122 [cs] doi:10.48550/arXiv.2403.15122

[3] 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.). doi:10.1007/978-3-642-25379-9_22

[4] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Integrated Formal Methods*, Nadia Polikarpova and Steve Schneider (Eds.). Springer International Publishing, Cham, 102–110. doi:10.1007/978-3-319-66845-1_7

[5] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission Accounting in Separation Logic. *SIGPLAN Not.* 40, 1 (Jan. 2005), 259–270. doi:10.1145/1047659.1040327

[6] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis*, Radhia Cousot (Ed.). Springer, Berlin, Heidelberg, 55–72. doi:10.1007/3-540-44898-5_4

[7] 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. doi:10.1145/3632902

[8] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C – A Software Analysis Perspective. In *SEFM (Lecture Notes in Computer Science, Vol. 7504)*. Springer, 233–247. doi:10.1007/978-3-642-33826-7_16

[9] 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. doi:10.1007/978-3-662-44202-9_9

[10] Thibault Dardinier, Peter Müller, and Alexander J. Summers. 2022. Fractional Resources in Unbounded Separation Logic. *Proc. ACM Program. Lang.* 6, OOPSLA2 (Oct. 2022), 163:1066–163:1092. doi:10.1145/3563326

[11] Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. 2025. Formal Foundations for Translational Separation Logic Verifiers. *Proc. ACM Program. Lang.* 9, POPL (Jan. 2025), 569–599. doi:10.1145/3704856

[12] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *ICFEM (Lecture Notes in Computer Science, Vol. 13478)*. Springer, 90–105. doi:10.1007/978-3-031-17244-1_6

[13] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer, Berlin, Heidelberg, 504–528. doi:10.1007/978-3-642-14107-2_24

[14] Marco Eilers and Peter Müller. 2018. Nagini: A Static Verifier for Python. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 596–603. doi:10.1007/978-3-319-96145-3_33

[15] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *ESOP (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 125–128. doi:10.1007/978-3-642-37036-6_8

[16] 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

[17] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, part i: a multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 927–942. doi:10.1145/3385412.3386014

[18] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local reasoning for storable locks and threads. In *Asian Symposium on Programming Languages And Systems*. Springer, 19–37. doi:10.1007/978-3-540-76637-7_3

[19] Ekanshdeep Gupta, Nisarg Patel, and Thomas Wies. 2025. Raven: An SMT-Based Concurrency Verifier. In *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 15931)*, Ruzica Piskac and Zvonimir Rakamaric (Eds.). Springer, 81–106. doi:10.1007/978-3-031-98668-0_4

[20] 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

[21] Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. 2015. Modular Termination Verification. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 664–688. doi:10.4230/LIPIcs.ECOOP.2015.664

[22] 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 (NFM) (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. doi:10.1007/978-3-642-20398-5_4

[23] Bart Jacobs, Frédéric Vogels, and Frank Piessens. 2015. Featherweight VeriFast. *Logical Methods in Computer Science* Volume 11, Issue 3 (Sept. 2015). doi:10.2168/LMCS-11(3:19)2015

[24] 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 *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. doi:10.1145/2676726.2676980

[25] Bernhard Kragl and Shaz Qadeer. 2021. The Civl Verifier. In *#PLACEHOLDER_PARENT_METADATA_VALUE#*. TU Wien Academic Press, 143–152. doi:10.34727/2021/isbn.978-3-85448-046-4_23

[26] Robbert Krebbers and Freek Wiedijk. 2013. Separation logic for non-local control flow and block scope variables. In *International Conference on Foundations of Software Science and Computational Structures*. Springer, 257–272. doi:10.1007/978-3-642-37075-5_17

[27] K. Rustan M. Leino. 2008. This Is Boogie 2. (June 2008). Available from https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml178.pdf.

[28] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, Berlin, Heidelberg, 348–370. doi:10.1007/978-3-642-17511-4_20

[29] K. Rustan M. Leino and Peter Müller. 2009. A Basis for Verifying Multi-threaded Programs. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer, Berlin, Heidelberg, 378–393. doi:10.1007/978-3-642-00590-9_27

[30] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115. doi:10.1145/1538788.1538814

[31] Hongyi Ling, Thibault Dardinier, Ellen Arlt, and Peter Müller. 2026. *Sound State Encodings in Translational Separation Logic Verifiers (Artifact).* doi:10.5281/zenodo.21509502

[32] Hongyi Ling, Thibault Dardinier, Ellen Arlt, and Peter Müller. 2026. Sound State Encodings in Translational Separation Logic Verifiers (Extended Version). arXiv:2603.20001 [cs.PL] https://arxiv.org/abs/2603.20001

[33] Andreas Lööw, Seung Hoon Park, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Opale Sjöstedt, and Philippa Gardner. 2025. Compositional Symbolic Execution for the Next 700 Memory Models. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 2815–2842. doi:10.1145/3763151

[34] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 827–850. doi:10.1007/978-3-030-81688-9_38

[35] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, Berlin, Heidelberg, 41–62. doi:10.1007/978-3-662-49122-5_2

[36] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. doi:10.1007/3-540-45949-9

[37] Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer, Berlin, Heidelberg, 49–67. doi:10.1007/978-3-540-28644-8_4

[38] 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. doi:10.1145/3656438

[39] 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. doi:10.1007/978-3-030-81688-9_33

[40] Willem Penninckx, Bart Jacobs, and Frank Piessens. 2015. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer, Berlin, Heidelberg, 158–182. doi:10.1007/978-3-662-46669-8_7

[41] João Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix Wolf, Marco Eilers, Christoph Sprenger, David Basin, Peter Müller, and Adrian Perrig. 2025. Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2025-11-22) *(CCS '25)*. Association for Computing Machinery, 1469–1483. doi:10.1145/3719027.3765104

[42] João C. Pereira, Isaac van Bakel, Patricia Firlejczyk, Marco Eilers, and Peter Müller. 2025. Modular Reasoning about Global Variables and Their Initialization. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 2310–2337. doi:10.1145/3763133

[43] J.C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. doi:10.1109/LICS.2002.1029817

[44] Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit Dynamic Frames. *ACM Trans. Program. Lang. Syst.* 34, 1 (May 2012), 2:1–2:58. doi:10.1145/2160910.2160911

[45] Simon Spies, Niklas Mück, Haoyi Zeng, Michael Sammler, Andrea Lattuada, Peter Müller, and Derek Dreyer. 2025. Destabilizing Iris. *Proc. ACM Program. Lang.* 9, PLDI (2025), 848–873. doi:10.1145/3729284

[46] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David Basin. 2020. Igloo: soundly linking compositional refinement and separation logic for distributed system verification. *Artifact for "Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification"* 4, OOPSLA (Nov. 2020), 152:1–152:31. doi:10.1145/3428220

[47] 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. doi:10.1007/S10009-020-00559-Y

[48] GitHub user: Aurel300. 2022. Shared reference unsoundness (issue #1189). https://github.com/viperproject/prusti-dev/issues/1189.

[49] GitHub user: cliu369. 2024. Assert(false) passing with Shared References (issue #1518). https://github.com/viperproject/prusti-dev/issues/1518.

[50] GitHub user: fpoli. 2024. Wrong encoding of signed discriminants in pure code (issue #1501). https://github.com/viperproject/prusti-dev/issues/1501.

[51] GitHub user: gewlar. 2026. Unsound behavior when casting to object (issue #281). https://github.com/marcoeilers/nagini/issues/281.

[52] GitHub user: marcoeilers. 2024. Unsoundness relating to static fields (issue #161). https://github.com/marcoeilers/nagini/issues/161.

[53] GitHub user: pieter-bos. 2021. Send/Recv is unsound where guard checking is incomplete (issue #703). https://github.com/utwente-fmt/vercors/issues/703.

[54] GitHub user: pieter-bos. 2023. SimplifyNestedQuantifiers makes an unsound rewrite (issue #955). https://github.com/utwente-fmt/vercors/issues/955.

[55] GitHub user: superaxander. 2025. Unsoundness due to violation of effective type rules (issue #1357). https://github.com/utwente-fmt/vercors/issues/1357.

[56] Viktor Vafeiadis. 2011. Concurrent Separation Logic and Operational Semantics. *Electronic Notes in Theoretical Computer Science* 276 (Sept. 2011), 335–351. doi:10.1016/j.entcs.2011.09.029

[57] 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.). doi:10.1145/2509136.2509532

[58] 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. doi:10.1007/978-3-540-95891-8_51

[59] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 367–379. doi:10.1007/978-3-030-81685-8_17

[60] 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. doi:10.1007/S10703-023-00427-W

[61] Junming Zhao, Miki Tanaka, Johannes Åman Pohjola, Alessandro Legnani, Tiana Tsang Ung, H. Truong, Tsun Wang Sau, Thomas Sewell, Rob Sison, Hira Syeda, Magnus Myreen, Michael Norrish, and Gernot Heiser. 2025. Verifying Device Drivers with Pancake. arXiv:2501.08249 [cs.PL] https://arxiv.org/abs/2501.08249

[62] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound Gradual Verification with Symbolic Execution. *Proc. ACM Program. Lang.* 8, POPL (Jan. 2024), 85:2547–85:2576. doi:10.1145/3632927

Received 2026-03-17; accepted 2026-08-06