Foundational Multi-Modal Program Verifiers

VLADIMIR GLADSHTEIN, National University of Singapore, Singapore
GEORGE PÎRLEA, National University of Singapore, Singapore
QIYUAN ZHAO, National University of Singapore, Singapore
VITALY KURIN*, Neapolis University Pafos, Cyprus
ILYA SERGEY, National University of Singapore, Singapore

Multi-modal program verification is a process of validating code against its specification using both dynamic and symbolic techniques, and proving its correctness by a combination of automated and interactive machine-assisted tools. In order to be trustworthy, such verification tools must themselves come with formal *soundness* proofs, establishing that *any* program verified in them against a certain specification does not violate the specification's statement when executed. Verification tools that are proven sound in a general-purpose proof assistant with a small trusted core are commonly referred to as *foundational.*

We present a framework that facilitates and streamlines construction of program verifiers that are both foundational and multi-modal. Our approach adopts the well-known idea of *monadic shallow embedding* of an *executable* program semantics into the programming language of a theorem prover based on higher-order logic, in our case, the Lean proof assistant. We provide a library of monad transformers for such semantics, encoding a variety of computational effects, including state, divergence, exceptions, and non-determinism. The key theoretical innovation of our work are *monad transformer algebras* that enable *automated derivation* of the respective sound verification condition generators. We show that proofs of the resulting verification conditions enjoy automation using off-the-shelf SMT solvers and allow for an interactive proof mode when automation fails. To demonstrate versatility of our framework, we instantiated it to embed two foundational multi-modal verifiers into Lean for reasoning about (1) distributed protocol safety and (2) Dafny-style specifications of imperative programs, and used them to mechanically verify a number of non-trivial case studies.

CCS Concepts: • **Software and its engineering** → **Formal software verification**.

Additional Key Words and Phrases: multi-modal verification, mechanised proofs, Lean, Dijkstra monads

ACM Reference Format:
Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. 2026. Foundational Multi-Modal Program Verifiers. *Proc. ACM Program. Lang.* 10, POPL, Article XX (January 2026), 31 pages. https://doi.org/10.1145/XXX

1 Introduction

The promise of formal software verification is to deliver programs that are rigorously proven to satisfy their ascribed specifications, eliminating any possibility of runtime errors. While the past two decades have seen a surge of large-scale formally verified systems, ranging from compilers [54, 63] and cryptographic libraries [30, 82] to operating systems [37, 51] and distributed consensus

*Work done during a research internship at National University of Singapore.

Authors' Contact Information: Vladimir Gladshtein, National University of Singapore, Singapore, vgladsht@comp.nus.edu.sg; George Pîrlea, National University of Singapore, Singapore, gpirlea@comp.nus.edu.sg; Qiyuan Zhao, National University of Singapore, Singapore, qiyuanz@comp.nus.edu.sg; Vitaly Kurin, Neapolis University Pafos, Cyprus, v.kurin@nup.ac.cy; Ilya Sergey, National University of Singapore, Singapore, ilya@nus.edu.sg.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/1-ARTXX
https://doi.org/10.1145/XXX

protocols [40, 101, 106], formal verification is far from being considered mainstream in software development. As noted by practitioners, the cost/benefit ratio of formally proving software systems correct often makes formal verification a poor choice for code quality assurance [26] in comparison with lightweight methods for bug finding, such as fuzz testing [12] and symbolic execution [16].

Multi-modal verification is a methodology that aims to lower the cost of adopting formal methods by combining lightweight and rigorous techniques for checking a program against its specification. Multi-modal verifiers allow one to run tests, perform symbolic checks, and prove correctness of a program using deductive reasoning. Since the latter task is by far the costliest in terms of time and intellectual burden, an ideal verifier should provide a suite of tools to discharge proof obligations for a program's correctness, through a combination of automated decision procedures and human-assisted proofs. As of today, several existing verification frameworks offer features for multi-modal verification, targeting both general-purpose programming models [43, 55, 71, 89, 98] and problem-specific modelling domains, such as distributed and security protocols [73, 74, 100]. However, to the best of our knowledge, no such tool come with formal machine-checked proofs of *their own soundness*: that is, none of them provide strong guarantees that the result of successful verification they report is consistent with the runtime behaviour of the respective program that has been verified in it.[1] In other words, such verifiers *can* miss bugs in programs.

A well-known methodology to engineer a formally sound verifier is to embed its modelling and specification languages into those of a general-purpose theorem prover, such as Rocq [86] or Lean [24]. This design shifts the trust from an implementation of the verifier to the implementation of the underlying prover, as long as the verifier's own correctness is established in the prover's logic. While those provers are also programs, they have a very small trusted core, and are generally believed to be bug-free. Verification frameworks built this way are called *foundational* and are typically employed in correctness-critical domains, such as verification of systems code [5, 17] and reasoning about programs with complex semantic features [38, 46, 91]. Implementing a foundational program verifier for a new programming language is typically a substantial effort on its own, and, while several attempts to automate foundational verification have been made recently [22, 33, 90, 94], we are not aware of any such tool providing a full spectrum of multi-modal verification features: testing, symbolic execution, and automated/interactive deductive proofs.

In this work, we present an approach for building foundational multi-modal program verifiers.

Challenges and key ideas. Any verification task starts from encoding a model of a program and its desired specification, so the *syntax*, in which the program and its specification are expressed, can have a large impact on the verifier's adoption. Our methodology for implementing verifiers is by *embedding* them into the Lean theorem prover and by making use of its support for *meta-programming* [80] to provide domain-specific syntax for the users. We aim to provide an experience where the user need not be proficient in Lean to be able to conduct their verification tasks, by employing their domain-specific knowledge, while implicitly relying on Lean's proving capabilities.

While the problem of providing user-friendly syntax can be solved with state-of-the-art meta-programming techniques, defining the *semantics* of a verifier's modelling language, which is both *executable* and *amenable to automated verification*, is the first conceptual challenge we have to solve. An appealing idea to encode such semantics is by "assembling" them from collections of *effects* that a language can express: state manipulation, exceptions, non-determinism, *etc*. An approach based on *Dijkstra monads* [96] provides an elegant theoretical foundation to *derive* principles for symbolic reasoning about programs with effects from their execution semantics. Unfortunately, the research on Dijkstra monads to date [4, 70] does not immediately provide a push-button solution

[1]This, of course, does not mean that those verification tools are not useful: they are typically rigorously tested, and soundness-compromising bugs in them are relatively rare, although definitely not non-existent [13].

for deriving a sound *automated verifier*, stopping just one step short of doing so. We identify this shortcoming and provide a solution to it by introducing a novel algebraic structure called *monad transformer algebras*. We show that a program semantics with execution-time effects that can be expressed as a composition of monad transformer algebras allows for automated derivation of a sound symbolic verifier that enjoys automation-friendly verification conditions.

The second conceptual challenge has to do with accommodating two styles of symbolic reasoning about code: symbolic execution and deductive verification. The former is more suitable for reliably identifying true bugs in programs without the need to prove anything [15], while the latter aims to show the absence of bugs at the expense of having to construct a proof. We reconcile them by observing that each one of them corresponds to one of two different ways of reasoning about non-determinism in a program's semantics: symbolic execution corresponds to taking *angelic* choices that exercise an execution that will likely result in a bug, while deductive verification considers *demonic* non-determinism, effectively checking that every execution satisfies a desired property [14]. We provide a reusable executable semantics for non-determinism and employ monad transformer algebras to derive verifiers suitable for both sound symbolic bug-finding and safety verification for potentially non-terminating, non-deterministic computations.

The last challenge we have to address is providing smooth language-agnostic integration between two styles of deductive program verification: *automated* and *user-assisted*. The former style is more popular in standalone *intrinsic* code verifiers, such as Dafny [58], Viper [76], and Verus [55], which require the user to annotate a program with the specification and invariants, emitting verification conditions (VCs) that are then discharged by an external automated solver. The latter style is predominant in foundational verifiers embedded into an interactive theorem prover that state the verification conditions *extrinsically* and use the underlying prover's interactive proof mode to discharge them [5, 44, 53]. Both styles have their merits: while the former greatly reduces the burden on the human prover, the latter provides more fine-grained control over the proof process. We adopt the intrinsic proof style, embedding our verification condition generators into Lean and developing language-independent automation techniques for proving VCs with the help of off-the-shelf solvers, such as Z3 [23] and cvc5 [8], as well as interactively. We show that our proofs often require no manual effort at all, while the ability to combine automated and interactive modes allows for verifying programs outside the reach of existing intrinsic automated verifiers.

We implement these ideas in Loom—a foundational framework for deriving verifiers embedded into Lean proof assistant. We showcase Loom by instantiating it to two realistic multi-modal verifiers, using the outlined above techniques in tandem to test and verify a series of case studies.

Contributions. In summary, our work makes the following contributions:

- Our main pragmatic contribution is Loom: a framework for embedding multi-modal verifiers into Lean with foundational end-to-end correctness guarantees and a good support for user-level automation. Loom comes with a library of shallowly-embedded computational effects, including state, divergence, and non-determinism, and allows for multiple styles of verification: testing, symbolic execution, and deductive proofs, both interactive and automated (Sec. 2).
- Our main theoretical contribution is a framework of *ordered monad (transformer) algebras*: an algebraic structure that streamlines deriving of correct-by-construction deductive verifiers from executable semantics represented by compositions of computational monads. Loom comes with a library of monad algebra instances for common effects that can be easily extended (Sec. 3).
- We demonstrate how to automate derivation of foundational deductive verifiers from monad algebra instances using Lean's type class resolution mechanism (Sec. 4).
- We show how to express divergence in our framework in a way that avoids coinductive definitions (Sec. 5) and present a monad transformer algebra for non-determinism (Sec. 6).

- We implemented two foundational multi-modal verifiers on top of Loom: (1) Veil 2.0, a complete overhaul of the Veil verifier for distributed protocols in first-order logic [83], now featuring executable semantics (Sec. 7), and (2) Velvet, a new Dafny-style verifier for an imperative language featuring arrays, unbounded loops, and the constrained random choice operator (Sec. 8). We used both verifiers to test and prove several non-trivial real-world case studies correct, highlighting the unique capabilities of multi-modal verification in a foundational proof assistant.

The entire development of this paper comes with end-to-end correctness proofs done in Lean.

2 Overview

We start by building the intuition for Loom's design and outlining the experience of using it. To do so, we will go through stages of developing a multi-modal verifier for Cashmere—a toy imperative WHILE-style language for implementing simple monetary operations, embedded into Lean.

2.1 Embedding Stateful Computations into Lean

Programs in Cashmere manipulate a simple state that contains a single mutable component representing the balance on a user's account (for demonstration purposes, let us forgo realism and assume the language only supports one user). We begin by implementing a function that withdraws a desired amount from a current balance. Its code is shown in Fig. 1a, and it is defined in a Lean monad `StateM`, which represents a stateful computation, first reading the current value of the user account's balance (line 3) and then updating it with the new amount (line 4).

Lean's meta-programming facilities make it easy to disguise the code in Fig. 1a, to make it look closer to familiar imperative code by adding syntax (`balance := ...`) for updating mutable state components, for **`returns`** clause to specify the return value, as well as **`require`**/**`ensures`** statements to specify its execution contract in terms of pre- and postconditions. The code in Fig. 1b macro-expands to the original code in Fig. 1a, which can be executed in Lean natively using the `StateM.run` function. Based on the user-provided **`ensures`** annotation, the definition in Fig. 1b also generates the corresponding correctness theorem for `withdraw`, which we will discuss next.

2.2 Specifying Stateful Computations with Loom

The correctness theorem generated by Loom for `withdraw` from its specification looks as follows:

$$\forall\ \mathsf{amount}\ b_{\mathrm{old}},\ \{\lambda b.\ b = b_{\mathrm{old}}\}\ \mathsf{withdraw\ amount}\ \{\lambda res\ b.\ b + \mathsf{amount} = b_{\mathrm{old}}\} \qquad (1)$$

The Hoare triple (1) is quantified over the input amount and the pre-defined *ghost* logical variable b_{old} capturing the initial balance. Its precondition is a predicate on the program state b, stating that its initial state is b_{old}. The postcondition is a predicate on the program result *res* (of type `Unit`) and the final state, also denoted as b. It states that the final balance state plus the input amount is equal to the initial balance (for now, we allow the balance to be negative). The program with **`require`**/**`ensures`** clauses is macro-expanded to a term `triple P c Q`, where `triple` is Loom's definition that can be used for computations `c` in any monad M that is equipped with an instance of `MAlgOrdered` type class (explained in Sec. 3.3). To instantiate this type class, one has to provide:

(1) A *complete lattice* L, to serve as the assertion language for computations in M, used to state pre- and postconditions. One can think of it as a type of propositions over the respective state.
(2) A *symbolic runner* function $\mu : M\ L \longrightarrow L$, which interprets M-computations ending with a postcondition in L as assertions in L. Specifically, μ "treats" assertions as values: it "runs" the computation `f`, returning an assertion that must hold in order for all its "resulting" assertions to hold true. One can think of μ as a weakest precondition predicate transformer *for a fixed postcondition*. It also provides meaning to the annotations such as **`invariant`**, **`assert`**, *etc.*

```
1 abbrev Bal := Int
2 def withdraw (amount : Nat) : StateM Bal Unit := do
3   balance ← get
4   set (balance - amount)
```

(a) Changing balance of the account

```
1 bdef withdraw (amount : Nat) returns Unit
2 ensures balance + amount = balanceOld do
3   balance := balance - amount
```

(b) Changing balance with macros

```
1 instance: MAlgOrdered (StateM Bal) (Bal → Prop) where
2   μ (f : Bal → (Bal → Prop) × Bal) :=
3     λ b =>
4       let (post, b') := f b
5       post b'
6   ... -- A proof that μ is monotone
```

(c) Embedding stateful computations

```
1 bdef withdraw (amounts : List Nat) returns Unit
2 ensures balance + amounts.sum = balanceOld do
3   let mut tmp := amounts
4   while tmp.nonEmpty
5   invariant balance    + amounts.sum =
6             balanceOld + tmp.sum do
7     let amount := tmp.head
8     balance := balance - amount
9     tmp := tmp.tail
```

(d) Session-based withdrawal loop

Fig. 1. Programming and verification in Cashmere

(3) A proof that μ is *monotone w.r.t.* the order on L. This requirement will be explained in Sec. 3.

Changing the shapes of M and L would require a different instance of `MAlgOrdered`. Luckily, Loom minimises such implementation overhead by providing a methodology for assembling computations modularly as a combination of monads and monad transformers including StateT, ReaderT, ExceptT, *etc*. That is, if the semantics of a computation can be expressed as a composition of these transformers [67], Loom will derive the corresponding instance of `MAlgOrdered` *automatically*.

Coming back to our example, the assertion language L for `StateM Bal` computations is just a type of predicates on the balance value `Bal → Prop`. It follows that μ has type `StateM Bal L → L`, which unfolds into `(Bal → (Bal → Prop) × Bal) → Bal → Prop`. Lines 3-5 of Fig. 1c show elements of its implementation. The function μ's first argument is `f : Bal → (Bal → Prop) × Bal`, which performs a stateful computation and returns both a poststate (`b'`) and an assertion about that poststate (`post`); μ runs `f` on the initial state `b` and returns its output assertion applied to its resulting state. Having provided the implementation of μ, together with the proof of its monotonicity *w.r.t.* implications on `Bal → Prop` (Sec. 3 will describe the general structure), Loom derives the weakest precondition transformer wp [25] for `StateM Bal` computations. The statement (1), thus, becomes

$$\forall\ \text{amount}\ b_{\text{old}}\ b,\ \ b = b_{\text{old}} \Rightarrow \text{wp (withdraw amount) } (\lambda res\ b'.\ b' + \text{amount} = b_{\text{old}})\ b \qquad (2)$$

For this example, wp(withdraw amount) *post* returns $\lambda\ b.post\ ()\ (b - \text{amount})$, meaning that the postcondition *post* will hold on a final state b if it holds on the computation's output $()$: Unit, and the state b − amount. After substituting this into the statement (2), the obtained verification condition (VC) can be discharged either via translation to SMT-LIB with one of existing Lean-SMT toolkits [75, 84], or via Lean's own tactics. To improve both automation and human-assisted proving experience, for more complicated VCs, Loom provides a tactic to split them into separate goals, where each goal corresponds to a proof supporting one **`ensures`** or **`invariant`** annotation. For this example, the proof of the VC can be obtained by running Lean's standard aesop tactic [68], thus, delivering an independently verifiable certificate of correctness in a form of Lean proof term.

2.3 Adding Non-Terminating Loops

Next, let us extend Cashmere with loops. This will make it possible to implement a procedure for handling multiple withdrawals in a single session by passing a list withdrawal amounts to `withdraw` (*cf.* Fig. 1d) decrementing the balance by the value of each of the list's elements in a loop (lines 4-9). An observant reader might notice that we do not supply an explicit termination measure for the loop—the mechanism of potentially non-terminating computations is enabled by Lean's machinery of partial fixpoint, which we will touch upon in more details in Sec. 5. To support reasoning about such partial loops, Loom implements the proof principles for partial correctness, wherein any postcondition holds true for a loop that does not terminate. By the virtue of shallow embedding into Lean, Loom supports the execution of divergent computations natively (*i.e.*, not by extraction, and linking with external code as done in Rocq [65]). In addition, Loom allows one to reason about non-terminating computations both intrinsically (via loop invariants) and extrinsically (via explicit theorems), disentangling proofs of partial and total correctness (*cf.* Sec. 2.4).

```
1 instance : MAlgOrdered (StateT Bal DivM)
2                          (Bal → Prop) where
3   μ (f : Bal → Option ((Bal → Prop) × Bal)) := λ b =>
4     match f b with
5     | some (post, b') => post b'
6     | none => True -- for partial correctness
7   ...
```

Fig. 2. Specifying divergent computations

A popular way to encode semantics of non-terminating computations is via coinductive definitions [62, 102], which, at the time of this writing, are missing from Lean. Instead of using coinduction, to encode divergence, we will change the underlying monad of our examples to `StateT Bal DivM Bal` where `StateT` is a state monad transformer [67], and `DivM` is an alias for `Option`. The semantic value of divergent computations will correspond to `none`. Although the assertion language L for `StateT Bal DivM` remains the same, we need to change the definition of μ and re-prove its properties, as shown at lines 3-6 of Fig. 2: now `f b` can return `none`, in which case we simply return `True`. The choice of the value for `none` is dictated by the notion of partial correctness; in Sec. 5, we will explain the choice, as well as its alternative, and will show how to avoid doing the monotonicity proofs altogether.

The property we want to verify is that the initial balance is the same as the final one plus a sum of withdrawn amounts. To support automated VC generation for new `withdraw` function, we need to add a **`invariant`** annotation to the loop. Loom will generate a VC for the desired property based on its **`ensures`** and **`invariant`** annotations and will dispatch it automatically with the help of `aesop`.

2.4 Proving Total Correctness

The obvious problem with partial correctness semantics is that any postcondition trivially holds for divergent computations. For instance, removing the last two lines of Fig. 1d, `withdraw` will make it run forever, but its VC will still be discharged successfully. Loom allows one to control the semantics of divergent computations by changing line 6 in Fig. 2 to `False`. In this case, the weakest precondition of a divergent computations would be `False`, so no postcondition could be proved for them. To enable an intrinsic proof of `withdraw` function, one would need to add an extra annotation with the decreasing measure after the **`invariant`** clause, such as **`decreasing`** `amounts.length`.

To accommodate both partial and total correctness proofs, we define `withdraw` using the `Option` monad, where it *might* diverge. A correctness proof of `withdraw` in the total semantics, thus, ensures that `withdraw` terminates, so Loom derives for it the following correctness statement:

$$\forall \text{ amounts } b_{\text{old}},\ \{\lambda b.\ b = b_{\text{old}}\}\ \text{ withdraw amounts }\ \{\lambda res\ b.\ b + \text{amounts.sum} = b_{\text{old}}\}$$

It implies that, for any `amounts`, the outcome of `withdraw` is always `some` (res, b), so that we can extract its result and state for the postcondition. We will show in Sec. 8 that, to get a total correctness proof for a program, one can first prove its partial correctness *w.r.t.* a desired property and then, separately, prove that `True` as a postcondition holds in the total semantics. Loom automatically combines these two proofs into a proof of total correctness *w.r.t.* a desired property.

2.5 Reasoning about Programs with Exceptions

Until this point, the type of Cashmere state was `Int`, and the programs did not enforce a clearly desirable property that the value of the balance always remains positive. To eliminate such scenarios in our running examples, we will change the implementation of `withdraw` to throw an exception if one is attempting to withdraw a larger value than the one of the current balance. The amended

```
 1  bdef withdraw (amounts : List Nat) returns Unit
 2  require balance ≥ amounts.sum
 3  ensures balance + amounts.sum = balanceOld do
 4    let mut tmp := amounts
 5    while tmp.nonEmpty
 6    invariant balance    + amounts.sum =
 7              balanceOld + tmp.sum
 8    decreasing tmp.length do
 9      let amount := tmp.head
10      if amount > balance then throw "Insufficient funds"
11      else balance := balance - amount
12      tmp := tmp.tail
```

(a) Withdrawal with exceptions

```
1  μ (f : ExceptT α BankM (Bal → Prop)) :=
2    μ (m := BankM) do
3      let ex ← (f : BankM (Except α (Bal → Prop)));
4      return ex.getD (λ _ => False)
```

(b) Semantic embedding of exceptions

```
 1  bdef withdraw returns (history : List Nat)
 2  require balance ≥ 0
 3  ensures balance + history.sum = balanceOld do
 4    let amounts :| amounts.sum ≤ balance
 5    let mut tmp := amounts
 6    while tmp.nonEmpty
 7    invariant balance    + amounts.sum =
 8              balanceOld + tmp.sum
 9    invariant balance ≥ tmp.sum
10    decreasing tmp.length do
11      let amount := tmp.head
12      if amount > balance then
13        throw "Insufficient funds"
14      else
15        balance := balance - amount
16      tmp := tmp.tail
17    return amounts
```

(c) Withdrawal with non-determinism

Fig. 3. Cashmere programs featuring exceptions and non-determinism

code is shown in Fig. 3a (lines 10-12). Note that now, to statically ensure that the exception is not thrown, we have added a precondition requiring that the current balance is greater than or equal to the sum of the amounts in the list passed as the argument (line 2).

To support exceptions in the executable semantics, we will have to adjust our computational monad once again, making it into `ExceptT String (StateT Bal DivM) Bal` where `ExceptT` is the exception monad transformer and `String` is the type of exceptions. Luckily, due to the similarity in the working of the state and exception monad transformers, this time, we do not need to fully redefine the semantic embedding. Instead, we can define the new version μ function by reusing the old one. Fig. 3b shows a definition of μ for `ExceptT String (StateT Bal DivM)`, where `StateT Bal DivM` is abbreviated as `BankM`. This definition immediately delegates to the old μ function for `BankM`, which in its turns expects an element of type `BankM (Bal → Prop)`. To construct the latter, we first trivially cast `f` to `BankM (Except String (Bal → Prop))` by unfolding the definition of `ExceptT`, and then extract its result using `Except.getD`, which retrieves an actual value of type `Bal → Prop` from `Except String (Bal → Prop)`, returning a default element (in this case, constant `(λ _ => False)`) if it is an exception. In general, Loom is capable to automatically derive an instance of `MAlgOrdered` for monads enhanced with exceptions, based on an "inner" instance and the instance for `ExceptT String`. Since line 4 of Fig. 3b returns `(λ _ => False)` for exception-throwing computations, the derived weakest precondition for `withdraw` will be just `False` for such executions. Hence, the specification ascribed in Fig. 3a ensures that, once verified, `withdraw` is exception-free.

2.6 Modelling Non-Determinism in Program Semantics

As another effect, Loom supports non-deterministic computations, which are useful, for instance, to model interactions with Input/Output. Continuing with our example, instead of passing a list of amounts to deduct from the account balance to `withdraw` as an argument, we can model it as a non-deterministic choice provided by the user interactively. Line 4 of Fig. 3c shows how that can be implemented in Cashmere by making use of Dafny-style *let-such-that* operator `:|` (also known as Hilbert's epsilon operator) [59]. The semantics of such a choice in our example can be summarised as "picking a random sequence of amounts to withdraw, such that the sum of the amounts does not exceed the current balance". The non-deterministically picked `amounts` also serves as the result of the whole procedure, which we name `history`, as it contains the latest history of withdrawals.

Loom derives VC to prove the new version of `withdraw` correct by assuming *demonic* semantics for the non-deterministic choice. That is, the correctness theorem for `withdraw` will ensure that its post-condition holds for *all* values of `amounts` satisfying the constraint at line 4. Furthermore, given a generator for values of type `List Nat`, Loom will provide a *sound* execution of `withdraw`, where it will attempt to generate `amounts` candidates until it finds one satisfying the `amounts.sum ≤ balance` condition. Sec. 6 describes the implementation of this feature in detail.

2.7 Symbolic Execution with Angelic Non-Determinism

In addition to modelling demonic choices for the sake of safety reasoning, Loom also allows to use *angelic* non-determinism to support symbolic execution of programs. As the last variation to our example, imagine that the `withdraw` implementation from Fig. 3c does not impose any constraints on the choice at line 4. In such a case, it would be nice to *prove* (in the sense of program logics for under-approximate reasoning [77, 107]) that the exception at line 10 may be thrown. To achieve this, we instruct Loom to use angelic non-determinism and treat exceptions as "success" (the exact mechanism will be shown in Sec. 4). With this semantics, we will try to prove `False` post-condition. Intuitively, verification of such goal can only succeed if an exception is thrown, because, in this case, the VC will ignore any postcondition and will just return `True`.

In the interest of space, we do not show the code of this example here, but it can be found in our Lean development, along with the Lean proof of its incorrectness.

2.8 Putting It All Together

We demonstrated how to implement a series of examples, building a Lean-embedded verifier for a custom effectful language using Loom (and a bit of Lean meta-programming). As of now, Loom comes with an extensible library of monad transformers for modelling a variety of computational effects, including StateT, ReaderT, ExceptT, NonDetT, DivM, and Gen; the last one is the Lean monad for random sampling [56] in the style of QuickCheck [19].

Some of Loom's supported effects include parameters that define their execution semantics and VC generation. For ExceptT with exceptions of type ε, one can specify how to treat exceptions: as failures, as successes, or by using a custom exception handler of type $\varepsilon \longrightarrow \mathsf{Prop}$. For DivM, one can control the symbolic treatment of diverging computations: partial or total. In the first case, one does not have to provide a decreasing measure; in the second, correctness proofs guarantee that the program always terminates. Finally, for NonDetT one can control how to treat non-deterministic choices: demonically or angelically. The former case corresponds to accounting for all possible non-deterministic choices, which is useful to prove program safety. The latter corresponds to symbolic model checking and is useful to prove the presence of a bug or reasoning about reachability.

In the next section, we explain the theoretical foundations of Loom and its mechanism of monad transformer algebras that enables automatically deriving VC generators for executable semantics.

3 Deriving Verifiers via Monad Transformer Algebras

At the heart of Loom's theoretical foundations is a subclass of Dijkstra (*i.e.*, specification-producing) monads [4, 70, 96] allowing for "push-button" derivation of generators of Floyd-Hoare-style verification conditions (VC) [31, 45], which often can be discharged using first-order logic solvers [8, 23]. In this section, we outline the necessary background on Dijkstra monads (Sec. 3.1), followed by a discussion of their pragmatic shortcomings for automatically deriving VC generators (Sec. 3.2). We then introduce additional restrictions on Dijkstra monads captured by the *ordered monad algebras* structure that makes it possible to automate the derivation of VC generators for the respective computation monads (Sec. 3.3). The section culminates with the novel idea of *monad transformer algebras*, extending the benefits of monad algebras to a large class of monad transformers, allowing for automated derivation of VC generators for semantics involving multiple effects (Sec. 3.4).

3.1 Background on Dijkstra Monads

Dijkstra monads [4, 70, 96] are a well-studied formalism for enabling deductive reasoning about effectful monadic computations. Ahman et al. have proposed an approach to derive, for a given monad M, which is used to perform computations with some effect (*e.g.*, state manipulation or exceptions), a corresponding *canonical specification* monad [4]. Such specification monads allow one to state logical specifications of monadic computations using a tailored version of Dijkstra's weakest precondition calculus [25], composing the specifications for program commands in a monadic style, mimicking the way ordinary monads implement "effect passing", with the ultimate goal to enable Hoare-style program verification [45]. In addition to providing a recipe to derive *canonical* specification monads for computational monads (which themselves can be assembled out of several monad transformers [67]), the work by Ahman et al. introduced a general approach for establishing soundness of a user-defined specification monad *w.r.t.* some computational monad.[2]

As an example, for the state monad StateM σ α, consider a specification monad StateW σ α, which is defined as $(\alpha \rightarrow \sigma \rightarrow \mathsf{Prop}) \rightarrow (\sigma \rightarrow \mathsf{Prop})$. The intuition behind this definition is that computations in StateW σ α correspond to *weakest precondition transformers.* That is, a value w of type StateW σ α encodes a specification of some computation done in StateM σ α, which, taking a desired post-condition *post* on its final state, returns the sufficient precondition w *post* to hold on the initial state. As any other monad, specification monad interface consists of the the implementations of functions (1) $\mathsf{pure} : \alpha \rightarrow W\alpha$ needed to inject pure values in monadic computations and (2) $\mathsf{bind} : W\alpha \rightarrow (\alpha \rightarrow W\beta) \rightarrow W\beta$ needed to compose computations (or, in this case, specifications). For StateW σ α, the standard definitions are pure $x \triangleq \lambda post.\ post\ x$ and bind $w\ f \triangleq \lambda post.\ w\ (\lambda res.\ f\ res\ post)$. The implementation of bind first obtains the weakest precondition of an application $f\ res$ with respect to a given postcondition *post*, and then uses it as a postcondition for w, to derive the weakest precondition of the composition.

The specifications in StateW are not yet connected to the computations in StateM. Such a connection is given by the morphism $\mathcal{WP}_{\mathsf{StateM}} : \mathsf{StateM}\ \sigma\ \alpha \rightarrow \mathsf{StateW}\ \sigma\ \alpha$ defined as follows:

$$\mathcal{WP}_{\mathsf{StateM}}\ (m : \mathsf{StateM}\ \sigma\ \alpha) \triangleq \lambda\ post\ (s : \sigma).\ \mathbf{let}\ (res, s') = (m\ s)\ \mathbf{in}\ post\ res\ s' \tag{3}$$

In plain words, we say that for a given post-condition *post*, $\mathcal{WP}_{\mathsf{StateM}}\ m\ post$ returns a precondition, which holds on an initial state s if and only if *post* holds on the result *res* of the computation $m\ s$ and its final state s'. Such transformation $\mathcal{WP}$ in general has the type $\forall\alpha, M\ \alpha \rightarrow W\ \alpha$ and defines a semantic mapping of a *whole* program (*i.e.*, $m : M$) to its specification (in the monad W) by "running" m via the call $m\ s$. To enable modular reasoning about computations in m, we should be able to derive the weakest precondition of a composite computation from its parts. The laws allowing us to do that for a computation monad M and a specification monad W are as follows:

$$\forall x,\ \mathcal{WP}\ (\mathsf{pure}_M\ x) = \mathsf{pure}_W\ x \tag{4}$$
$$\forall m\ f,\ \mathcal{WP}\ (\mathsf{bind}_M\ m\ f) = \mathsf{bind}_W\ (\mathcal{WP}\ m)\ (\lambda r.\ \mathcal{WP}\ (f\ r)) \tag{5}$$

To give more intuition on how these laws are related to compositional program verification, recall that bind $m\ f$ usually encodes a *sequential composition* of a command m and its continuation f. In this context, the law (5) states that to get its specification, we just need to compose the specification of m with the specification of f applied to m's result r. The only "basic" rules that need to be proven are those specific to a monadic computation M in question, such as get/set for StateM.

An additional property of $\mathcal{WP}$ that is essential for scalable verification in practice is *monotonicity.* For StateM, this property means that for a stateful computation m, and any two post-conditions

[2]This is useful for computation monads such as List, which cannot be represented as a monad transformer [2] and, thus, does not yield with a canonical instance of a Dijkstra monad following Ahman et al.'s methodology.

$post_1$ and $post_2$, if $post_1\ r\ s$ implies $post_2\ r\ s$ for any r and any state s (*i.e.*, $post_1$ refines $post_2$) then

$$\forall s,\ \mathcal{WP}_{\mathsf{StateM}}\ m\ post_1\ s \Rightarrow \mathcal{WP}_{\mathsf{StateM}}\ m\ post_2\ s \tag{6}$$

This property is crucial for reusability of specifications and their proofs. For example, proving $\mathcal{WP}\ m\ (\lambda s.\ s = 5)$ means that we also have a proof of $\mathcal{WP}\ m\ (\lambda s.\ s > 0)$.

To conclude our tour of a specification monad for StateM, we show that $\mathcal{WP}_{\mathsf{StateM}}$ defined via (3) can be used to define program correctness statements in a form of Floyd-Hoare triples:

$$\{pre\}\ m\ \{post\} \triangleq \forall s.\ pre\ s \Rightarrow \mathcal{WP}_{\mathsf{StateM}}\ m\ post\ s \tag{7}$$

The above means that *pre* implies the weakest precondition of m *w.r.t.* postcondition *post*.

Before we move on, note that this recipe to define Hoare triples is somewhat bespoke for state monad. Specifically, the form of StateW as well as definitions (7) and (3) crucially rely on the structure of StateM. To wit, for the List monad, the specification monad would look like $(\alpha \to \mathsf{Prop}) \to \mathsf{Prop}$: it would feature no "state" component σ, and for a given postcondition *post*, the weakest precondition would have to hold for *all* (or *at least one*, depending on the desired verification style—more on that in Sec. 6) elements of the list to satisfy *post*. The Hoare triple definition would have to be adapted accordingly. It is natural to wonder: (a) how do specification monads W for a given computation monad M would look in general and (b) whether it is always possible to define a Hoare triple in a chosen specification monad? To answer these questions, let us discuss the metatheory of Dijkstra monads and the pragmatic limitations caused by its generality.

3.2 Specification Monads and Hoare Triples

To tackle the concerns raised above, let us first identify a common pattern in the specification monads we have seen so far: $\mathsf{StateW}\ \sigma\ \alpha \triangleq (\alpha \to \sigma \to \mathsf{Prop}) \to (\sigma \to \mathsf{Prop})$ for the state monad, and $\mathsf{ListW}\ \alpha \triangleq (\alpha \to \mathsf{Prop}) \to \mathsf{Prop}$ for the list monad. Clearly, both such monads resemble the *continuation monad* $\mathsf{Cont}\ L\ \alpha \triangleq (\alpha \to L) \to L$, where, in the former case L is $\sigma \to \mathsf{Prop}$, and in the latter, L is just Prop. To capture this pattern, the general type of specification monads we will be considering in this work will be $\mathsf{Cont}\ L\ \alpha$, and we will refer to the type L as *assertion language*. The intuition behind identifying an assertion language for a monad M is simple: it is just a type of assertions, which one can state about computations done by elements of M. An assertion language defines the type of pre- and postconditions for monadic computations, and one should expect an assertion language itself to come with some properties that make it useful for writing specifications. In this work, we require each assertion language L to be a *complete lattice*, which guarantees that its elements (*i.e.*, assertions) come equipped with many useful operations, such as meet and join for disjunction and conjunction, top and bottom for truth and falsity, supremum and infimum for existential and universal quantifiers, and partial order for implication.

Now, given a specification monad $\mathsf{Cont}\ L\ \alpha$ for a computation monad M, with $\mathcal{WP} : M\ \alpha \to \mathsf{Cont}\ L\ \alpha$ being a weakest precondition morphism (so far, bespoke for L and M) and $\leq$ being a partial order on L, a Hoare triple for a monadic computation $m : M$ can be defined as follows:

$$\{pre\}\ m\ \{post\} \triangleq pre \leq \mathcal{WP}\ m\ post \tag{8}$$

The following property of $\mathcal{WP}$ generalises the monotonicity requirement (6) for StateM:

$$\forall m\ (post_1\ post_2 : \alpha \to L), (\forall (x : \alpha), post_1\ x \leq post_2\ x) \Rightarrow \mathcal{WP}\ m\ post_1 \leq \mathcal{WP}\ m\ post_2 \tag{9}$$

Notice that by imposing the lattice structure on L, we manage to generalise (and, thus, automate) almost all steps from Sec. 3.1 towards defining Hoare triples for a given computational monad M and a specification language L. The recipe boils down to using $\mathsf{Cont}\ L\ \alpha$ as a specification monad for M, defining triples as done in (8). The only wrinkle is that one still has to explicitly provide the weakest precondition morphism $\mathcal{WP} : M\ \alpha \to \mathsf{Cont}\ L\ \alpha$: remember, for StateM monad the definition has been provided in an ad-hoc way by (3), and it would look differently for ListM.

Indeed, for many sensible choices of M and L, it is possible to define $\mathcal{WP}$ explicitly and prove its properties (4), (5), and (9), to obtain the suitable Hoare logic. However, the lack of automation in this aspect is what stands in the way of "push button" generation of sound program verifiers for languages with arbitrary executable semantics expressed as a composition of monadic effects.[3]

This brings us to the main conceptual contributions of this work: (1) identifying an algebraic structure that relates a computation of type M and an assertion language L, and allows for deriving a morphism $\mathcal{WP}$ for M and L, which satisfies the properties (4), (5), and (9), thus, automatically delivering a sound Hoare logic (and the respective VC generator) for M, and (2) extending this structure to executable semantics that are defined as combinations of multiple effects.

3.3 Monad Algebras

For a given monad M, there is a one-to-one correspondence between weakest precondition morphisms $\mathcal{WP} : M\ \alpha \to \mathsf{Cont}\ L\ \alpha$ satisfying properties (4) and (5) (but not necessary (9)) and structures called *monad algebras* that relate M with an assertion language L:[4]

Definition 3.1 (Monad Algebra). For a given computational monad M we say that type L and a morphism $\mu : M\ L \to L$ form a *monad algebra* if the following two laws hold:

(1) for any $p : L$, $\mu\ (\mathsf{pure}\ p) = p$
(2) for any $m : M\ \alpha$, $f, g : \alpha \to M\ L$, if $\forall x, \mu(f\ x) = \mu(g\ x)$ then $\mu(m \gg= f) = \mu(m \gg= g)$

In this context, the type L is referred to as *monad algebra object.*

Before providing an intuition for this definition, we enhance the notion of a monad algebra to extend this one-to-one correspondence to weakest precondition morphisms that are monotone (9):

Definition 3.2 (Ordered Monad Algebra). A complete lattice L with a partial order $\leq$ and a morphism $\mu : M\ L \to L$ form an *ordered monad algebra* for a monad M if the following laws hold:

(1) for any $p : L$, $\mu\ (\mathsf{pure}\ p) = p$
(2) for any $m : M\ \alpha$, $f, g : \alpha \to M\ L$, if $\forall x, \mu(f\ x) \leq \mu(g\ x)$ then $\mu(m \gg= f) \leq \mu(m \gg= g)$

The only difference with Definition 3.1 is that we replace "=" with "≤" in the second law. From here on, we will only consider ordered monad algebras, referring to them as just monad algebras. Intuitively, the first law of Definition 3.2 states that the "symbolic run" of computation simply returning an assertion p is just p, while the second law captures the monotonicity property of μ. That is, strengthening some part of the symbolic run of a program (by, *e.g.*, imposing a stronger assertion at the end) should only strengthen the overall outcome of a symbolic run.

Let us show how to derive a morphism $\mathcal{WP}$ from a given μ of a monad algebra:

$$\mathcal{WP} : M\ \alpha \to (\alpha \to L) \to L \triangleq \lambda(m : M\ \alpha)\ (post : \alpha \to L).\ \mu\ (post \mathbin{\langle\$\rangle} m) \tag{10}$$

The $\langle\$\rangle$ operator above is a monadic mapping of type $(\alpha \to \beta) \to M\ \alpha \to M\ \beta$. To understand the intuition behind (10), note that $post \mathbin{\langle\$\rangle} m$ is essentially the same monadic computation as m but returning an assertion $post\ a$ instead of an "ordinary" value a. Applying μ to $(post \mathbin{\langle\$\rangle} m)$ results in a precondition that must hold before running $post \mathbin{\langle\$\rangle} m$ for it to return a true assertion.

For $\mathcal{WP}$ defined this way, we can state the following theorem (which we proved in Lean):

[3]In this work we deliberately do *not* account for all possible monadic implementations of effects, as some effects can have multiple representations. For instance, non-determinism can be represented via backtracking, lists, trees, *etc* [3, 42, 49].
[4]This observation is discussed in Sec. 4.4 of the paper by Maillard et al. [70], who derive Dijkstra monads using a more general mechanism of *effect observations* [47], which does not guarantee the monotonicity property (9).

Theorem 3.3 (Properties of $\mathcal{WP}$ derived from Ordered Monad Algebra). *If an assertion language L and a morphism $\mu : M\ L \to L$ form an ordered monad algebra for a monad M, then $\mathcal{WP}$ derived using* (10) *satisfies the properties* (4), (5), *and* (9).

As mentioned in Sec. 3.1, $\mathcal{WP}$ with properties (4), (5), and (9) immediately provide rules to compose the weakest preconditions of basic operations into specifications of composite programs. That said, we have not yet discussed how to derive $\mathcal{WP}$s for basic operations for a given monad (*e.g.*, get/set of StateM). While we do not automate this aspect in this work, these derivations can be usually done mostly mechanically by unfolding the respective definitions of μ and $\mathcal{WP}$.

As an example, consider the derivation of $\mathcal{WP}$ for StateM's get operation, *i.e.*, a computation of type StateM $\sigma\ \sigma$ which returns an underlying state. By definition, we unfold get into $\lambda s.\ (s, s)$: for an initial state s, get's outcome is this exact state s, hence the following derivation:

$$\mathcal{WP}_{\mathsf{StateM}\ \sigma}\ \mathsf{get}\ post\ s = \mu_{\mathsf{StateM}\ \sigma}(post \mathbin{\langle\$\rangle} \mathsf{get})\ s \tag{11}$$

$$= \mu_{\mathsf{StateM}\ \sigma}(post \mathbin{\langle\$\rangle} \lambda s.\ (s, s))\ s \tag{12}$$

$$= \mu_{\mathsf{StateM}\ \sigma}(\lambda s.\ (post\ s, s))\ s = post\ s\ s \tag{13}$$

The last line (13) of the derivation above can be validated by unfolding the definition of $\mu_{\mathsf{StateM}\ \sigma}$.

Following the outlined methodology, obtaining a Hoare-style verifier and a respective VC generator for a computation monad M boils down to (a) selecting an assertion language L and a morphism μ (both are usually relatively straightforward, as shown in Fig. 1c), (b) proving the laws stated in Definition 3.2, and (c) deriving the definitions of $\mathcal{WP}$ for all operations specific to the monad. However, interesting programs rarely involve just one effect (*e.g.*, just state), so next, we will show how to extend this approach to semantics of computations combining multiple monads.

3.4 Monad Transformer Algebras

In the previous section, we have discussed a recipe to derive weakest precondition calculi for specification monads of the form Cont $L\ \alpha$ by leveraging the mechanism of monad algebras. In this section, we will address the challenge of deriving monad algebras for composite monads.

For example, assume we have defined a monad algebra instance for a monad M, and now we want to add a state component to it by making use of the StateT monad transformer. Do we have to define an instance of monad algebra for StateT $\sigma\ M\ \alpha$ from scratch, proving all the required properties, or is there a compositional way to do so, allowing for proof reuse? Below, we provide an approach to achieve the latter, by introducing a monad algebra analogue for monad transformers.

The main idea is simple: for a given monad transformer T, which "adds" a particular effect, let us define a monad algebra instance of this transformer as it were applied to an arbitrary monad M, which itself already came with a monad algebra instance formed by L and μ. We illustrate how it can be done for StateT. First, assuming that assertion language for M is L, we have to define an assertion language for StateT $\sigma\ M$. For the assertions on stateful computations in StateT $\sigma\ M$, we want to be able to state everything we could have stated for M, and additionally reason about state. This suggests to define the new assertion language as $\sigma \to L$. Now, we need to define $\mu_{\mathsf{StateT}\ \sigma\ M}$ of type (StateT $\sigma\ M\ (\sigma \to L)) \to (\sigma \to L)$ by making use of $\mu_M : M\ L \to L$. Before going through its definition, remember that StateT $\sigma\ M\ \alpha$ is defined as $\sigma \to M(\alpha \times \sigma)$: given an initial state of type σ, computations in StateT $\sigma\ M\ \alpha$ yield monadic executions in M carrying result α and an updated state. Therefore, for m of type $\sigma \to M((\sigma \to L) \times \sigma)$, we define

$$\mu_{\mathsf{StateT}\ \sigma\ M}\ m \triangleq \lambda(s : \sigma).\ \mu_M(apply \mathbin{\langle\$\rangle} (m\ s)) \tag{14}$$

Above, $apply : (\alpha \to \beta) \times \alpha \to \beta$ is a function that applies the first element of a pair to the second one. In the definition (14), we first run m on an initial state s to get a monadic computation $m\ s$ of type $M((\sigma \to L) \times \sigma)$. Next, we apply the predicate it returns (of type $(\sigma \to L)$) to the updated state (of type σ) via "*apply* $\langle\$\rangle$". The result of this operation (*apply* $\langle\$\rangle$ $(m\ s)$) has type $M\ L$, so at the end we can apply μ_M to obtain an assertion of type L.

As we prove in our Lean development, by assuming that μ_M satisfies laws from Definition 3.2, we can derive the same laws for $\mu_{\mathsf{StateM}\ \sigma}$. That means, when composing StateT with other monads, one would not have to redefine and reprove everything from scratch, getting a correct-by-construction ordered monad algebra. Furthermore, defining μ for transformers this way makes it possible to derive the weakest preconditions of transformer-specific basic computations by abstracting over the structure of monad algebras of underlying monads! For example, for the get operation of type StateT $\sigma\ M\ \sigma$, defined as $\lambda s.\ \mathsf{pure}(s, s)$, the weakest precondition is as follows:

$$\mathcal{WP}\ \mathsf{get}\ post\ s = \mu_{\mathsf{StateT}\ \sigma\ M}(post\ \langle\$\rangle\ \mathsf{get})\ s \tag{15}$$

$$= \mu_{\mathsf{StateT}\ \sigma\ M}(post\ \langle\$\rangle\ \lambda s.\ \mathsf{pure}(s, s))\ s = \mu_{\mathsf{StateT}\ \sigma\ M}(\lambda s.\ \mathsf{pure}(post\ s, s))\ s \tag{16}$$

$$= \mu_M(apply\ \langle\$\rangle\ \mathsf{pure}(post\ s, s)) = \mu_M(\mathsf{pure}(post\ s\ s)) = post\ s\ s \tag{17}$$

In the derivation above, steps (15) and (16) correspond to steps (11) and (13) from the $\mathcal{WP}$ derivation of get in Sec. 3.3, while step (17) follows from the law $\mu_M(\mathsf{pure}\ p) = p$ from Definition 3.2.

Now, we are ready for the first attempt towards defining a monad algebra analogue for monad transformers. If a monad algebra is defined for a monad algebra object (*cf.* Definition 3.1) represented by Lean type L, then, for a monad transformer which transforms a monad into another, such an object should be correspondingly a type constructor, or more specifically, an *endofunctor* on Lean types (*i.e.*, a mapping Type $\to$ Type that preserves identity and function composition). For instance, for StateT $\sigma\ M$, we defined its monad algebra object to be $F_{\mathsf{StateT}\ \sigma\ M} \triangleq \lambda L.\sigma \to L$, where L is a monad algebra object for M. This suggests to define a *monad transformer algebra* for a monad transformer T as an endofunctor F, such that for any monad algebra formed by L and $\mu : M\ L \to L$, there exists a function $F\mu : T\ M(F\ L) \to F\ L$ satisfying the monad algebra laws. In this work, we impose one extra constraint on F, which is useful for deriving verifiers automatically.

To identify the problem with the definition suggested above, imagine you are working with the monad $T\ M$ defined as a transformer T applied to a monad M. Assume that the monad M comes with a basic operation $q : M\ \alpha$ (think get from StateM) and the transformer T provides an operation $t : T\ M\ \beta$, which is agnostic to the underlying monad M. With the definition above, if our library contains the $\mathcal{WP}$ derived for t, we can directly use it in our proofs, as this definition is agnostic to any particular monad M. However, this does not apply to the computation q. Even if our library contains the $\mathcal{WP}$ derived for M, a monad $T\ M$ will not use q directly, but will rather run $\mathsf{lift}_T\ q$ instead (where lift_T is a monad morphism for lifting computations from an arbitrary monad M to $T\ M$, supplied as a part of the interface of the monad transformer T).

If we want to enable *automated* derivation of $\mathcal{WP}$s for lifted computations from M irrespective of the shape of the composite monad, we need to impose a law on the definition of a monad transformer algebra that states how to *commute* lift and $\mathcal{WP}$. Let us first show how this law can be stated in terms of lift and μ, and then derive a correspondent law for $\mathcal{WP}$. To develop the necessary intuition, consider our running example of StateT. We can show that for $m : M(\sigma \to L)$

$$\mu_{\mathsf{StateT}\ \sigma\ M}(\mathsf{lift}_{\mathsf{StateT}\ \sigma\ M}\ m) = \lambda(s : \sigma).\ \mu_M(m\ \langle\&\rangle\ s) \tag{18}$$

Above, $\langle\&\rangle$ is a function of a type $M\ (\sigma \to L) \to \sigma \to M\ L$.[5] In plain words, if we have a lifted monadic computation m of type $M\ (\sigma \to L)$, then this computation clearly does not affect the state introduced by the state transformer. So, in order to eliminate the part of m's outcome responsible for state ($\sigma \to -$ part in $\sigma \to L$), we simply need to apply this outcome assertion to the *initial*

[5]Here, $\langle\&\rangle$ is defined as $\lambda\ m\ s.(\lambda(f : \sigma \to L).f\ s)\ \langle\$\rangle\ m$. It only requires M to be a functor.

state s (since this state is not modified by m's computation). This is done via m <&> s. The result of this expression has type $M\ L$, so we can apply μ_M to it to get a value of type L.

The definition (18) is what we would like to have in the general case, as it expresses the outcome of the symbolic run μ_{TM} of a T-lifted computation from M in terms of μ_M. To make it hold for arbitrary monad transformers, not just StateT, let us understand the demonstrated trick with <&>: using the function $\lambda s.\ m$ <&> s, we have turned $m : M(\sigma \to L)$ into a function of a type $\sigma \to M\ L$. Abstracting over the functor $\sigma \to -$ (which is specific to StateT) and replacing it with an abstract F, we realise that there should be a way to turn $m : M(F\ L)$ into a value of $F(M\ L)$. In other words, there should exist a so-called *distributive-law* between F and M.

At the time of this writing, there was no known general recipe to derive such laws between arbitrary endofunctors and monads [109]. However, in all our examples, endofunctors F are *representable*: they are of the form $\alpha \to -$ for some type α (such functors are called *Hom-functor* and denoted $\mathsf{Hom}(\alpha, -)$). Luckily, in this case one can define a general notion of <&> of type $M(F\ L) \to F(M\ L)$ by exercising the trick we did with StateT verbatim. First, we get a representation of F as $\mathsf{Hom}(\alpha, -)$ turning $M(F\ L)$ into $M(\alpha \to L)$, then we apply <&> to get $\alpha \to (M\ L)$, and finally, we "fold" the representation of F back to get $F(M\ L)$. We will abbreviate these steps as $\mathsf{distr}_F : M(F\ L) \to F(M\ L)$. We are now ready to introduce *monad transformer algebras.*

Definition 3.4 (Monad Transformer Algebra). Assume T is a monad transformer and F is a representable endofunctor. We say that F is a monad transformer algebra over T if

(1) for any monad algebra formed by L and μ, there exists a function $F\mu : T\ M(F\ L) \to F\ L$, such that $F\ L$ and $F\mu$ form a monad algebra over $T\ M$, and
(2) for such $F\mu$ and any $m : M(F\ L)$, $F\mu(\mathsf{lift}_T\ m) = \mu_M$ <$> $(\mathsf{distr}_F\ m)$.

Thanks to the second law, we can derive a desired property for $\mathcal{WP}$ of lifted computations:

Theorem 3.5 ($\mathcal{WP}$ for lifted computations). *Assume a monad transformer T and an endofunctor F form a monad transformer algebra. Assume also a monad M and L form an (ordered) monad algebra. Then for any $m : M\ \alpha$ and $post : \alpha \to F\ L$*

$$\mathcal{WP}_{TM}(\mathsf{lift}_{TM}\ m)\ post = (\mathcal{WP}_M\ m) \mathbin{\texttt{<\$>}} (\mathsf{distr}_F\ post) \tag{19}$$

A pragmatically-minded reader might wonder why calculating the right-hand side of (19) is simpler than implementing its left-hand side explicitly. Note that the type of *post* can be written as $\mathsf{Hom}(\alpha, -)(F\ L)$, so distr_F, according to its definition, will turn it into $F(\mathsf{Hom}(\alpha, -)(L)) \triangleq F(\alpha \to L)$. Now, as our F is representable (*i.e.*, it is $\mathsf{Hom}(\beta, -)$ for some β), it should be clear that the application of distr_F corresponds to just swapping arguments in $post : \alpha \to (\beta \to L)$, returning a value of type $\beta \to (\alpha \to L)$. This form of the assertion is now "passed" (via <$>) to the "inner" weakest precondition transformer $\mathcal{WP}_M\ m$, resulting in the outcome of type $\beta \to L$. This operation of pushing the assertions to the right specification monad can be effectively automated, yielding a push-button derivation of $\mathcal{WP}$ for lifted monadic computations.

Our framework Loom comes with instances of monad transformer algebras for monad transformers encoding common effects, such as state (both mutable and immutable), exceptions, and several others. A user of Loom willing to implement their own program verifier only has to choose a composition of transformers capturing their executable semantics to enjoy a sound VC generator derived automatically. One can also extend Loom with new monad transformers to support additional effects by providing suitable monad transformer algebra instances. In the next sections we will discuss the Lean-powered automation provided by Loom for deriving verifiers (Sec. 4) and describe Loom support for two important effects: divergence (Sec. 5) and non-determinism (Sec. 6).

```
class MAlgOrdered (l : outParam (Type v))
  [Monad m] [CompleteLattice l] where
  μ : m l → l
  μ_ord_pure : ∀ l, μ (pure l) = l
  μ_ord_bind {α : Type v} :
    ∀ (f g : α → m l), μ ∘ f ≤ μ ∘ g →
    ∀ x : m α, μ (x >>= f) ≤ μ (x >>= g)
```

Fig. 4. Ordered monad algebra type class

```
class MAlgLift (m : semiOutParam (Type u → Type v))
  (l : semiOutParam (Type u))
  [Monad m] [CompleteLattice l] [MAlgOrdered m l]
  (n : (Type u → Type w)) (k : outParam (Type u))
  [Monad n] [CompleteLattice k] [MAlgOrdered n k]
  [MonadLiftT m n] where
    [cl : LogicLift l k]
    wp_lift (x : m α) : wp (liftM n x) post = liftM (wp x) post
```

Fig. 5. A type class for monad transformer algebras

4 Encoding and Automating Loom Meta-Theory with Lean Type Classes

In this section, we discuss elements of our Lean library enabling automated verifier derivation.

Inferring assertion languages. Fig. 4 shows an encoding of the ordered monad algebra as a type class `MAlgOrdered` paraphrasing Definition 3.2 in Lean. This class requires an assertion language type `l` to be a complete lattice, and the type `m` should be a monad, *i.e.*, it should have `pure` and `bind` functions. The only unusual bit in our encoding is the annotation `outParam` of the type class argument `l`, which means that, even if the type of the assertion language is unspecified, Lean will try to infer it automatically based on the type class instances available in the context. To explain the true utility of `outParam`, assume we have a computation of type `StateM Int Unit` and want to get its weakest precondition by passing it to the respective `wp` function. Remember that the `wp` function has the following type for a computation monad `m` and an assertion language `l` (Sec. 3.2):

```
∀ {m : Type u -> Type v} {α l : Type u} [MAlgOrdered m l], m α → (α → l) → l
```

When we apply `wp` to `c : StateM Int`, Lean is able to automatically infer its arguments `m` and α from the type of `c`, but type `l` is not known. However, thanks to the fact that `l` is marked as an `outParam`, it can be *synthesised* from the `MAlgOrdered` instance for `StateM` automatically.

Scoped monad algebra instances. For extra flexibility, Loom makes it easy to control semantics of the weakest precondition calculi for the same monad by providing multiple scoped instances of `MAlgOrdered` for it in different namespaces. For instance, for `DivM` we have two instances with `True` and `False` values for `none` (*cf.* Sec. 2.3-2.4). One of them is defined in the Lean namespace `PartialCorrectness` and the other one in `TotalCorrectness`. Each instance is marked as **`scoped`**, meaning that they are only available in the respective namespaces and do not conflict with each other. As an example, to get a total, demonic semantics treating exceptions as success, one needs to open `TotalCorrectness`, `DemonicChoice` and `ExceptSuccess` namespaces.

Representing monad transformer algebras. Fig. 5 shows the Lean definition of a monad transformer algebra, which is quite different from what that of Definition 3.4. We encode it this way because Lean does not have a type class for monad transformers. To make class resolution more ergonomic and efficient, Lean encodes monad transformers as a type class for *monad lifting*:

```
class MonadLift (m : Type u → Type v) (n : Type u → Type w) where ...
```

This class specifies how to embed a monad `m` into a monad `n`. To follow this idiom in Loom, we implemented a type class `MAlgLift`, asserting that an ordered monad algebra for `m` and `l` can be lifted into an ordered monad algebra for `n` and `k`. By default, `MAlgLift` gives us (1) `LogicLift l k` representing the language `k` as α `→ l` for some α, and (2) `wp_lift` lemma, which is the Lean encoding of the equality (19). The `wp_lift` lemma relies on the `LogicLift` class, which provides the `liftM` function to lift `wp x : Cont l` α to `Cont k` α, corresponding to distr_F from Definition 3.4. Even though `MAlgLift` requires supplying a proof of the property of `wp`, but not of μ, we define an instance that derives the proof of `wp_lift` from `m`'s μ and its properties. Given an arbitrary computation in `m`, Loom's **`derive_lifted_wp`** command lifts its definition to the respective target monad `n`, provided a suitable instance of `MAlgLift` (which is usually derived automatically). For example, given a lemma specifying the weakest precondition for `get : StateT σ m σ`, for an arbitrary monad `m`

```
#derive_lifted_wp (σ: Type u) for (get : StateT σ m σ) as n σ
```

derives the weakest precondition for `get` function lifted into the monad `n` supporting stateful computations, so that the VC generator automatically produces VCs for `get` in the monad `n`.

5 Monad Algebra for Divergence

In this section, we outline the mechanism to define partial functions in Lean, and explain how to link it to the notion of monad algebras (Sec. 3.3) for reasoning about partial correctness in the presence of loops. Crucial to our goals will be a monad algebra instance for the Option monad, which is used in Lean to model divergence. For this monad, one can set the assertion language L to be Prop, $\mu(\text{some } p)$ to be p, and $\mu(\text{none})$ to be True or False, depending on the desired verification style: the former defines partial correctness, while the latter corresponds to total correctness.

5.1 Divergent Computations in Lean

Lean allows one to define generally-recursive functions inside a class of monads that come with a *chain-complete partial order* (CCPO) on the type of their computations.

Definition 5.1 (Chain-Complete Partial Order). A partial order $\leq$ on a type A is called *chain-complete* if for any chain $a_1 \leq a_2 \leq \dots$ in A there exists the least upper bound $\bigsqcup_i a_i$ in A.

Definition 5.2 (Chain-Complete Monad). A monad M is called *chain-complete* if it provides a chain-complete partial order on $M\ \alpha$ for any result type α.

For general recursion, it is additionally required that bind is monotone with respect to this CCPO:

Definition 5.3 (Monotonicity of bind*).* A chain-complete monad M is called *monotone* if for any $m_1, m_2 : M\ \alpha$, $f_1, f_2 : \alpha \to M\ \beta$, if $m_1 \leq m_2$ and $\forall x, f_1\ x \leq f_2\ x$ then $m_1 \gg= f_1 \leq m_2 \gg= f_2$.

With the Definitions 5.2 and 5.3 holding for a monad M, Lean defines generally-recursive monadic computations in M following a Knaster-Tarski-style construction [1, 10]. A trivial instance of a chain-complete monotone monad is Option. In this case, CCPO is defined as an order that only asserts that none $\leq$ some x for any x. Moreover, the Lean standard library establishes that if M is a chain-complete monotone monad and is transformed with one of the standard transformers, such as StateT, ExceptT, ReaderT, *etc*, the resulting monad remains chain-complete and monotone. Given such a monad and a function $f : M\ \alpha \to M\ \alpha$ (*i.e.*, function which takes the recursive call as an argument), Lean defines its fixpoint as $\bigsqcup_{x \in f^{\uparrow *}} x$, where $f^{\uparrow *} : M\ \alpha \to \text{Prop}$ (pronounced "f iterated") is an inductively defined smallest set such that (1) for the bottom element $\bot$ of a corresponding CCPO, $f\ \bot \in f^{\uparrow *}$, (2) for each $x \in f^{\uparrow *}$, $f\ x \in f^{\uparrow *}$, and (3) for each chain $c : M\ \alpha \to \text{Prop}$ such that $c \subseteq f^{\uparrow *}$, the least upper bound of c is also in $f^{\uparrow *}$.

With these amenities, Lean can define general loops (including `while`-loop) for a chain-complete monotone monad M without having to provide a termination measure. Loom exploits this feature, providing a general iteration operator iter [102, Sec. 3.2], which can be used to express many different kinds of loops, including `while`, `for`, `do-while`, *etc*. The definition of iter is parametric in an underlying monad M and has the type $\alpha \to (\alpha \to M(\alpha + \beta)) \to M\ \beta$. Intuitively, the output type of the iter-loop body can be either α or β, which is modelled by taking a tagged union of those types, denoted as +. The type α is ascribed to a value that is being computed by the each iteration of the loop and passed to the next iteration. The type β corresponds to a value indicating termination of the loop: if the result of the loop body is $b : \beta$, then the loop terminates. Clearly, no one guarantees that the body of the iter-loop will ever return a value of type β, hence this loop

$$\frac{\forall a,\ \{\mathsf{inv}(\mathsf{inl}\ a)\}\ m\ \{ab : \alpha + \beta,\ \mathsf{inv}(ab)\}}{\{\mathsf{inv}(\mathsf{inl}\ a_0)\}\ \mathsf{iter}\ a_0\ m\ \{b,\ \mathsf{inv}(\mathsf{inr}\ b)\}}$$

(a) Partial iteration rule

$$\frac{\forall a,\ \{\mathsf{inv}(\mathsf{inl}\ a)\}\ m\ \{ab : \alpha + \beta,\ \mathsf{inv}(ab) \wedge \ulcorner ab < a \urcorner\}}{\{\mathsf{inv}(\mathsf{inl}\ a_0)\}\ \mathsf{iter}\ a_0\ m\ \{b,\ \mathsf{inv}(\mathsf{inr}\ b)\}}$$

(b) Total iteration rule

Fig. 6. Total and partial iteration rules

might diverge. Even though Lean can execute such an operator natively, it does not provide us a standard invariant-based reasoning mechanism to reason about its safety. In the next section, we will define such a mechanism by harnessing the structure of monad algebras.

5.2 Reasoning about Partial and Total Correctness with Loops

Loom's general Hoare-style rules for iter are depicted at Fig. 6. Both of those rules rely on a traditional loop invariant inv. To match the type of iter discussed above, the invariant inv has the type $\alpha + \beta \longrightarrow L$, with L denoting assertion language associated with the monad M. Intuitively, $\mathsf{inv} : \alpha + \beta \longrightarrow L$ bundles together two functions: $\mathsf{inv}_{\mathsf{inl}} : \alpha \longrightarrow L$ and $\mathsf{inv}_{\mathsf{inr}} : \beta \longrightarrow L$. The former is used to express an invariant which must hold on the result computed by each iteration of the loop, and the latter expresses the condition which must hold upon the loop's termination. The rule depicted at Fig. 6a expresses the partial correctness property of the iter operator. If the invariant holds on the initial value $a_0 : \alpha$, injected into $\alpha + \beta$ via inl, we want to prove that after executing the iter-loop, the invariant will hold on the final loop result $b : \beta$, injected into $\alpha + \beta$ via inr. This rule corresponds to the partial semantics of iter, as it does not feature any condition to enforce its termination. The total correctness rule (Fig. 6b) is similar, but it also has a condition to enforce the loop termination. This condition is expressed via $\ulcorner ab < a \urcorner$ assertion, where $\ulcorner \cdot \urcorner$ is a notation to inject pure (*i.e.*, effect-agnostic) propositions into L. It means that the loop result $ab : \alpha + \beta$ is less than a if it is of a form $\mathsf{inl}\ a'$ and is vacuously true otherwise.

It turns out, the second rule is a valid rule for an arbitrary monad algebra where one can define the iter operator. However, for the first rule to be sound, an extra relation should hold on the definition of $\mathcal{WP}_M$ operator and the CCPO of M. For example, let us take $M = \mathsf{Option}$, applying iter to a body implemented as $m \triangleq \lambda a.\ \mathsf{pure}(\mathsf{inl}\ a)$, a function with no effect, and always returning result of type α. As in this case we will never get a result of type β, iter will diverge. Therefore, the rule's precondition holds for a trivially true invariant. At the same time, $\mathsf{iter}\ a_0\ m$ is none, so if we take $\mu_M(\mathsf{none}) = \mathsf{False}$, the respective $\mathcal{WP}_M(\mathsf{iter}\ a_0\ m)$ will return False, and the rule's conclusion will not hold. To resolve this unsoundness, we introduce *partial monad algebras*.

Definition 5.4 (Partial Monad Algebra). Assume M is a chain-complete monad. A monad algebra formed by L and $\mu : M\ L \longrightarrow L$ is called *partial* if for any chain $c \subseteq M\ \alpha$, $\bigwedge_{m \in c} \mathcal{WP}_M(m)\ post \leq \mathcal{WP}_M\left(\bigsqcup_{m \in c} m\right)\ post$ holds, where $\wedge$ is the meet (*e.g.*, conjunction) on the assertion type L.

To understand the meaning of this definition, assume c is a chain representing the set of iterations of a function f. Definition 5.4 states that if a precondition *pre* implies all the weakest preconditions $\mathcal{WP}(m)$ for every m in the chain c, then it also implies the weakest precondition of the least upper bound of the chain, which coincides with the least fixpoint of f. In other words, to prove a property about the fixpoint of f, it is enough to prove it for each iteration of f.

To understand why Option with $\mu(\mathsf{none}) = \mathsf{False}$ is not a partial monad algebra, assume that c is an empty chain. Then the left-hand side of the inequality in Definition 5.4 is a conjunction over an empty set, *i.e.*, True, and its right-hand side is the weakest precondition of the least upper bound of the empty chain, *i.e.*, False. As a valid example of a partial monad algebra, we can take the Option monad with $\mu(\mathsf{none}) \triangleq \mathsf{True}$. Moreover, for each monad transformer shipped with

```
inductive NonDetT M α where
  | pure : α → NonDetT M α
  | vis {β} : M β → (β → NonDetT M α) → NonDetT M α
  | pick (τ) : (τ → Prop) → (τ → NonDetT M α) → NonDetT M α
```

(a) A monad transformer for non-determinism

```
def μ (x : NonDetT M L) : L := match x with
  | pure p => p
  | vis x cont => μM ((λ a => μ (cont a)) <$> x)
  | pick τ p cont => ⨅ a ∈ p, μ (cont a)
```

(b) A monad algebra instance for NonDetT

Fig. 7. Nondeterministic monad transformer and its monad algebra instance

Loom, we have also proven that it preserves the partial monad algebra structure. That is, for a set of standard transformers T, we prove that if F forms a monad transformer algebra on T (Sec. 3.4), and L comes from a partial monad algebra on M, then $F\ L$ is a partial monad algebra for $T\ M$. In practice, it means that we provide rules for partial correctness triples for all compositions of supported effects defined as stacked transformers with the Option monad "at the bottom".

6 Specifying and Executing Non-Deterministic Computations

This section describes Loom's monad transformer for non-deterministic computations NonDetT mentioned in Sec. 2.6-2.7. First, we will show how to define a monad transformer algebra instance for NonDetT, and then discuss the execution semantics for this monad transformer.

6.1 Monad Transformer Algebra for Non-Determinism

We define a monad transformer for non-determinism using a variant of a so-called *program monad*, whose simplified version is shown in Fig. 7a [64]. The first two constructors are analogous to the definition of the *Interaction Tree* structure [102], and correspond to the pure and bind operations. The third constructor encodes a *Hilbert's epsilon operator* [34], whose operational meaning corresponds to non-deterministically picking an element of type τ satisfying the predicate p.

To implement an instance of a monad transformer algebra for NonDetT, recall that a monad algebra for some monad M is formed by the ordered assertion language $(L, \Rightarrow)$ and the "symbolic run" function $\mu : M\ L \longrightarrow L$. What would be a monad algebra for NonDetT M? Intuitively, for a non-deterministic computation that ends with some assertion from L, we would like this assertion to be true for *all* or *at least one* of possible outcomes of the computation, depending on the verification style one wants to use—so called *demonic* or *angelic* non-determinism [14]. This suggests that the assertion language for NonDetT M should be the same as that for M, and the semantics for a program would be expressed as a *conjunction*/*disjunction* of the assertions for each outcome.

The function μ implementing this intuition is depicted in Fig. 7b. For a pure assertion p : L, it simply returns p. For a vis of x : M β and cont : $\beta \longrightarrow$ NonDetT M L, the definition first recursively applies μ to cont obtaining a predicate of type $\beta \longrightarrow$ L. Next, this predicate is applied to x via <$> to get an element of type M L. Since we assumed that M already comes with monad algebra instance, we can apply the respective μ_{M} extracted from this instance to get an element of type L. For a non-deterministic choice pick, the definition takes an infimum $\bigsqcap$ of *all* semantics of $\mu(\text{cont}\ a)$ for each a from p, for which we require L to be a complete lattice. This definition corresponds to a *demonic choice* semantics for non-determinism. In our framework, we also provide an instance for an *angelic choice* semantics of NonDetT, which is obtained by taking the supremum at the definition of pick Fig. 7b. In the interest of space, in this section we only discuss the former.

For the type τ and a predicate p : $\tau \longrightarrow$ Prop, one can define the semantics of the Hilbert's epsilon operator pickSuchThat (denoted as x :| p in the language of Dafny verifier [59]) as follows:

$$\text{pickSuchThat}\ \tau\ \text{p} \triangleq \text{NonDetT.pick}\ \tau\ \text{p}\ \text{NonDetT.pure} \quad (20)$$

```
inductive Extract : NonDetT M α → Type where
  | pure : ∀ x : α, Extract (NonDetT.pure x)
  | vis {β} (x cont) : (∀ y : β, Extract (cont y)) →
        Extract (NonDetT.vis x cont)
  | pick (τ p cont) [Findable τ p] :
    (∀ t : τ, Extract (cont t)) →
        Extract (NonDetT.pick τ p cont)
```

(a) Choices for non-deterministic computations

```
def NonDetT.run (x : NonDetT M α) : Extract x → M α
  | pure x => pure_M x
  | vis x cont contEx => x >>= (λ a =>
     NonDetT.run (cont a) (contEx a))
  | pick τ p cont contEx => match find τ p with
     | some x => NonDetT.run (cont x) (contEx x)
     | none => ⊥
```

(b) Running computations in NonDetT

Fig. 8. Auxiliary definitions to run NonDetT

6.2 Executing Non-Deterministic Computations

Let us show how to soundly execute computations in NonDetT for demonic choice and the partial correctness semantics. Our accompanying formalisation provides implementations for other choices of semantics. We will show how to build a function NonDetT.run : NonDetT M $\alpha \to$ M α for any monad M equipped with a partial monad algebra instance (Sec. 5), such that the semantics of this obtained computation in M α will *refine* the semantics of the original computation in NonDetT M α (*i.e.*, will produce one of the possible outcomes captured by NonDetT M α).

The execution semantics for the first two constructors of NonDetT follow those of ITrees [102]: NonDetT.pure can be executed into pure operator from the underlying monad M, and NonDetT.vis can be executed using bind. To "execute" a non-deterministic choice operator, we must assume additional structure on the type τ and predicate p. To do so, we adopt the ideas on executing Hilbert's choice operator in Dafny [59], parametrising NonDetT.run by a *witness* of a (possibly partial) function that picks an element from τ satisfying a predicate p. We represent such witness as the inductive type Extract outlined in Fig. 8a. This definition recurses over the NonDetT.vis constructor, and for each NonDetT.pick constructor requires an instance of a Findable type class, provided by Loom. The most essential component of this type class is find : Option τ. Intuitively, if find returns a value x, then p should hold for x. Formally, this is captured by the type class field

$$\text{find\_spec} : \forall x : \tau, \text{find } \tau \text{ p} = \text{some x} \Rightarrow \text{p x} \tag{21}$$

Our implementation can infer an instance of this type class at each call to pickSuchThat, provided that τ is finite and p is a decidable predicate. In this case, find enumerates all elements of τ until it finds one which satisfies p. Assuming an instance of Findable, we can define a function NonDetT.run depicted at Fig. 8b. This function recurses over NonDetT.vis constructor, calling find for each NonDetT.pick constructor to pick an element from τ satisfying p. If that call returns value x, then we call NonDetT.run recursively via cont, otherwise we return a divergence value $\bot$.

We have proven the following soundness theorem for the outlined execution semantics in Lean:

Theorem 6.1 (Soundness of NonDetT.run). *If* M *has a partial monad algebra instance, then for any* c : NonDetT $M\ \alpha$, ex : Extract c*, and suitable pre-/post-conditions, the following holds:*

$$\{pre\}\ c\ \{post\} \Rightarrow \{pre\}\ \text{NonDetT.run}\ c\ ex\ \{post\}$$

If a monad M does not come with a partial monad algebra instance, then for NonDetT.run to be sound, we also need to ensure that each reachable call to a non-deterministic choice is *realisable*: if a program point where it makes a non-deterministic choice with a predicate p is reachable, this predicate should hold for some value. To formalise the intuition behind "reachability" in a monadic computation, Loom provides a method to derive a *weakest liberal precondition* calculus from the specific class of monad algebras; its details can be found in our Lean code.

7 Combining Runtime and Deductive Verification of Distributed Protocols

We used Loom to redefine and enhance the semantic foundations of the recently released Veil verifier [83], producing its new version Veil 2.0. Veil is an open-source multi-modal verification

```
def VeilM (m : Mode) (ρ σ α : Type) := NonDetT (ReaderT ρ (StateT σ (ExceptT ExId DivM))) α
def VeilM.succeedsWhenIgnoring (ex : Set ExId) (act : VeilM m ρ σ α) (pre : ρ → σ → Prop) :=
  [IgnoreEx ex|triple pre act (λ _ => ⊤)]
def VeilM.meetsSpecificationIfSuccessful act pre post := [DemonSucc|triple pre act post]
def VeilM.toTwoState act : ρ → σ → σ → Prop :=
  λ r₀ s₀ s₁ => [AngelFail|triple (λ r s => r = r₀ ∧ s = s₀) act (λ _ r s => r = r₀ ∧ s = s₁)]
```

Fig. 9. The Veil monad and definitions used for verification and model checking

framework embedded into Lean, which supports both automated and interactive verification of transition systems, with a focus on verifying distributed protocols. It features a simple imperative language, inspired by Ivy's RML [78], for users to specify initial states and protocol transitions, and a declarative language for describing safety properties and (finite) system traces, respectively.

The typical verification workflow in Veil is to first (*i*) specify the transition system and state its safety properties, then (*ii*) use Veil's SMT-based symbolic bounded model checking (BMC) to ensure the specification is not vacuous (*i.e.*, it admits non-trivial execution traces) and that the desired safety properties are not trivially violated (up to some small execution depth), and finally, once some initial trust in the correctness of the specification is thus built, to (*iii*) iteratively discover an inductive invariant that is a sufficient condition for the desired safety properties.

The Veil 2.0 benchmark suite contains 17 specifications of 15 different distributed protocols (two protocols are verified both in a decidable fragment, and separately, in general first-order logic).

7.1 Angelic Non-Determinism and the Semantics Zoo in Veil

Non-determinism features prominently in Veil specifications, particularly to abstract away implementation details or avoid formulations that would push the specification outside the decidable fragment. For instance, when specifying a consensus protocol, rather than encoding the length of a replica's log as a first-order **`function`** in Veil, it is preferable (to maintain decidability) to encode it as a **`relation`** with a coherence assumption, *i.e.*, a partial function. The value can then be retrieved using the pickSuchThat operator, as described in Sec. 6.1, without introducing quantifier alternation in the specification. This is an instance of non-deterministic choice. Another use-case, for angelic non-determinism, is symbolic bounded model checking, and in particular, checking that traces of particular shapes are admitted by the specification, *e.g.*, "is there a way to pick transition parameters such that transition A happens, followed by any two transitions, then followed by B?".

One of the major limitations in the original implementation of Veil was its approach towards such angelic non-determinism. Before we ported Veil to use Loom-provided semantics, its imperative actions were monadic programs elaborated in a *hard-coded* weakest-precondition monad with *demonic* choice and treated *exceptions as failures* (for all choices which satisfy the assumptions made about them, the transition terminates without an exception being thrown, and the post-condition holds).[6] This treatment is appropriate for verification, *e.g.*, for proving the safety of protocols, but is awkward for the kind of symbolic model checking described above. Specifically, the problem is that this kind of model checking relies on *angelic* choice and *exception as failure* semantics: a transition can occur if there is a way to choose its parameters such that all assumptions made about them hold and no exception is thrown. However, only *angelic* and *exception as success* semantics could be derived from Veil's original semantics (by negating the postcondition, and then negating the obtained weakest precondition). As such, Veil's symbolic execution was, as initially implemented, sound only for actions which *never threw exceptions*.

[6]Non-termination was not a concern in the original Veil, as it did not feature loops.

To address this limitation, as well as other shortcomings of Veil,[7] we completely overhauled its semantics using Loom. Veil actions now elaborate into programs in the `VeilM` monad, whose definition in terms of Loom monad transformers is shown in Fig. 9. Concretely, a Veil 2.0 action is a monadic computation that has read-only access to the **`immutable`** background theory of its specification (`ReaderT` ρ), can read and write to the **`mutable`** state (`StateT` σ), can raise exceptions and diverge (`ExceptT ExId DivM`), can perform non-deterministic choices (`NonDetT`), and returns a value of type α. The same definition can now be interpreted with different semantics. The most general of these semantics is `IgnoreEx`, which uses *demonic* choice and is parametrised by a set of exception IDs which result in success (*i.e.*, the set of permitted exceptions, which are "ignored"). If this set `ex` is `λ _ => ⊤`, then this corresponds to *demonic* choice and *exception as success* semantics (`DemonSucc`), which we use to check that Veil 2.0 actions preserve invariants. Setting `ex` to `λ _ => ⊥` leads to `DemonFail` semantics, *i.e.*, Veil's original semantics. More interestingly, if we set `ex` to `λ e => ¬ e` and check `IgnoreEx` for all exception IDs, we also obtain `DemonFail` semantics, but with the model returned by the SMT solver containing the ID of an exception which can be thrown. To obtain *all* assertions which can be violated in an action, it suffices to run this check in a loop, excluding previously seen exception IDs.

Veil 2.0 checks for each action both that (a) it does not throw any exceptions (assuming the invariant holds in the pre-state) and that (b) if it does not throw any exceptions, it preserves the invariant. If the (a) check fails, the failing assertion is highlighted to the user in the IDE. Using Loom's meta-theory, (a) and (b) together can be shown to imply the `DemonFail` semantics. Finally, the `AngelFail` semantics used to represent two-state transitions for symbolic model checking can also be derived from `IgnoreEx` via the double-negation trick used in the original Veil, but this time with respect to `DemonSucc` semantics. To summarise, Veil 2.0 actions can seamlessly be interpreted under any desired semantics, whilst running weakest precondition generation only once (for `IgnoreEx`), and *deriving* the VCs in other semantics by simple rewrites enabled by Loom's meta-theory.

Finally, as explained in Sec. 6.2, Veil 2.0 also has runtime execution semantics, including for actions with non-determinism. In the next section, we show how this proved useful.

7.2 Case Study: NOPaxos and Runtime Testing

To demonstrate how Veil 2.0 benefits from the enhancements enabled by Loom, as a new case study, we present a simplified version of NOPaxos consensus protocol [66]. NOPaxos, short for Network-Ordered Paxos, is a distributed protocol for state-machine replication, designed to operate in data-centres where the network provides an *ordered unreliable multicast* (OUM) primitive implemented by software-defined switches. The goal of the protocol is to coordinate multiple *replicas* to agree on a dynamically growing *log* of values, where each value is a client-issued request. The OUM tags every client request with a unique sequence number, and as such, the replicas only need to coordinate on whether to include or not include a particular request in their log (the channel is unreliable, so requests may not reach all replicas), but not on the order in which to include them. The *leader*, a distinguished replica, directs the coordination as the protocol executes. The original NOPaxos includes two subprotocols: view change and synchronisation. Our simplified model of the protocol omits these components, so we consider only the case with a fixed leader.

To specify a system in Veil, one begins by defining its type parameters and the global protocol state. Fig. 10a shows selected type and state declarations for NOPaxos in Veil: `replica` and `value` are the types of replicas and values, respectively, and `seq_t` is a type that abstracts natural numbers, retaining only the assumption that they encode a total order. The protocol state is divided

[7] *E.g.*, the immutability of the background theory was enforced using a syntactic check. Now it is enforced in the type.

```
type replica
type value
type seq_t
instantiate seq : TotalOrderWithZero seq_t
immutable individual leader : replica
relation r_log_len : replica → seq_t → Bool
relation r_log : replica → seq_t → value → Bool
```

(a) Type and state declarations of NOPaxos

```
procedure log_append (r : replica) (v : value) = {
  let len :| r_log_len r len
  let next_len ← succ len
  r_log r next_len v := true
  r_log_len r I := decide (I = next_len)
  return next_len
}
```

(b) One subprocedure of NOPaxos

Fig. 10. Extracts from the formalisation of NOPaxos in Veil

into *immutable* and *mutable* components, which represent background assumptions and evolving state, respectively. Given the fixed-leader assumption, the leader is declared as an **`immutable individual`**, where **`individual`** corresponds to a constant in first-order logic. The log maintained by each replica is **`mutable`**. For the sake of decidability, we avoid modeling it as a concrete sequence and instead use two relational abstractions: `r_log_len r i` asserts that replica `r`'s log has length `i`, while `r_log r i v` states that the $\mathtt{i}^{\text{th}}$ value in `r`'s log is `v`. Fig. 10b shows a subprocedure in NOPaxos that appends a value `v` to replica `r`'s log. It first retrieves the current log length `len` of `r` using pickSuchThat, then calls another subprocedure `succ` (definition omitted) to obtain `next_len`, the successor of `len`. The procedure then sets the $\mathtt{len}^{\text{th}}$ entry of `r`'s log to `v` and updates the length.

We ported NOPaxos to Veil based on an existing specification in Ivy [104] and followed the methodology described in Sec. 7. The existing specification already had an inductive invariant, which we ported over, with it verifying successfully. Moreover, we verified with symbolic model checking (now proven sound) that the protocol is not vacuous, *i.e.*, that it admits executions in which values are committed to the log. As a final experiment, we implemented a randomised simulation framework for Veil protocols as a monadic program and verified its soundness in Loom itself, proving that it produces only traces that are admitted by the specification. By inspecting the execution traces produced by the simulator, we noticed that one NOPaxos transition (`handle_gap_commit_reply`, which the leader runs after it does not receive a client request, decides to include a no-op in the log, and confirms this with the replicas) appeared to *never be taken*. And indeed, inspecting the relevant action's code, we confirmed that one of its **`require`** statements was always false. We fixed this action along with the affected invariant clauses and several related actions, and ultimately certified the safety property of the corrected specification of NOPaxos.

We could have, in principle, caught this error with symbolic model checking. In practice, we did not because of its slowness—using the SMT solver as a glorified execution engine is inefficient. Now that Loom provides us with concrete execution semantics, Veil users can do much better.

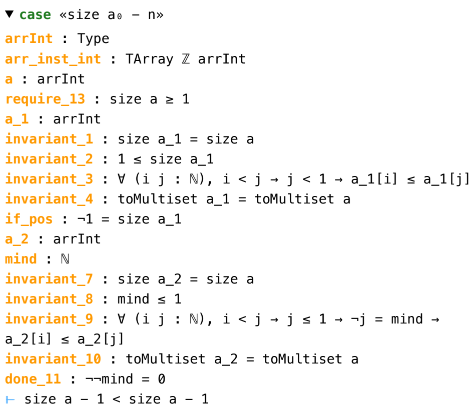
8 Combining Automated and Interactive Proofs in a Dafny-Style Verifier

As another case study for Loom, we have implemented Velvet: an embedding of a Dafny-style verifier [58] into Lean. Velvet allows one to verify stateful, possibly non-terminating programs with non-deterministic choices, manipulating arrays and algebraic data types. In addition to the deductive verifier, we have implemented support for QuickCheck-style property-based testing [19] for Velvet programs, so that one can test a program's specification before proving it.

The main advantage of Velvet over Dafny is that the former allows to combine automated and interactive proofs within the same verifier. Using Velvet, we have verified 10 case studies: in-place insertion sort, linear and binary search for square and cube roots calculations, and multiple computations on sparse matrices. While doing so, we have discovered two distinct ways in which Dafny-style intrinsic verification can benefit from Lean's proof mode. First we show how one can combine automated proofs of different semantic properties such as partial functional correctness and termination in Velvet to obtain a stronger specification Sec. 8.1. Second, we show how Velvet

```
1  method insertionSort (mut a: arrInt) return (u: Unit)
2  require 1 ≤ size a
3  ensures ∀ i j, i ≤ j < size a → aNew[i] ≤ aNew[j]
4  ensures toMultiset a = toMultiset aNew do
5    let a₀ := a
6    let mut n := 1
7    while n ≠ size a
8    invariant n ≤ size a
9    invariant ∀ i j, i < j < n → a[i] ≤ a[j]
10   invariant toMultiset a = toMultiset a₀
11   decreasing size a₀ - n do
12     let mut mind := n
13     while mind ≠ 0
14     invariant mind ≤ n
15     invariant ∀ i j, i < j ≤ n ∧ j ≠ mind → a[i] ≤ a[j]
16     invariant toMultiset a = toMultiset a₀
17     decreasing mind do
18       if a[mind] < a[mind - 1] then
19           swap a (mind - 1) mind
20       mind := mind - 1
21     n := n + 1
22  prove_correct insertionSort by loom_solve
```

Fig. 11. Velvet code and proof of insertion sort



```
▼ case «size a₀ - n»
arrInt : Type
arr_inst_int : TArray ℤ arrInt
a : arrInt
require_13 : size a ≥ 1
a_1 : arrInt
invariant_1 : size a_1 = size a
invariant_2 : 1 ≤ size a_1
invariant_3 : ∀ (i j : ℕ), i < j → j < 1 → a_1[i] ≤ a_1[j]
invariant_4 : toMultiset a_1 = toMultiset a
if_pos : ¬1 = size a_1
a_2 : arrInt
mind : ℕ
invariant_7 : size a_2 = size a
invariant_8 : mind ≤ 1
invariant_9 : ∀ (i j : ℕ), i < j → j ≤ 1 → ¬j = mind →
a_2[i] ≤ a_2[j]
invariant_10 : toMultiset a_2 = toMultiset a
done_11 : ¬¬mind = 0
⊢ size a - 1 < size a - 1
```

Fig. 12. Lean InfoView for a failed proof goal

can be used to combine automated and interactive proof modes to reason about programs featuring non SMT-friendly mathematical specifications as well as loops with complex invariants Sec. 8.2.

8.1 Proving Partial and Total Correctness of Insertion Sort

To account for diverging and non-deterministic computations, the Velvet computational monad is defined as VelvetM $\triangleq$ NondetT DivM. State mutability is modelled by piggy-backing on Lean's **let mut** syntax, which is a part of Lean's support for do-notation [97]. Unlike vanilla Lean, Velvet also allows to modify local mutable variables by passing them to function calls.

Fig. 11 shows an implementation, specification, and a proof of insertion sort in Velvet. Mutable method parameters passed by references are marked as **mut**. This is needed because Velvet operates only with Lean types and, hence, does not distinguish between mutable and pure data structures, such as arrays and sequences in Dafny. Velvet only allows to pass distinct identifiers for mutable parameters to a method's call, similarly to Dafny's `requires a != b` annotations, to avoid problematic aliasing. In Fig. 11, the parameter a is ascribed the abstract type `arrInt`, rather than Lean native type of integer arrays. This because Velvet sends its VCs to SMT via Lean-Auto [84], which currently does not support Lean arrays natively, so we provide the required properties via the `arrInt` "interface". Velvet makes this subtle difference transparent for the user, by providing all the relevant array notations for `arrInt`, as well as the proof that Lean arrays satisfy all its properties, so that the user can pass regular Lean arrays anywhere `arrInt` is expected.

Before attempting any verification, Velvet allows one to subject a method to property-based testing. For this, it provides the `derive_tester_for` command, which takes the name of a method and produces a property-based tester based on its specification. Provided a generator for the method's input values that satisfy the `require` predicate (which should be decidable),[8] the synthesised tester will pass them to the method and will run it natively, checking that the postcondition holds. If, *e.g.*, we change the postcondition at the line 3 of Fig. 11 to `∀ i j, i < j < size a → aNew[i] < aNew[j]`, we will get a counterexample such as `[9, -7, 9, -3]`, containing repeating values in the array.

To aid intrinsic verification, the Velvet implementation of insertion sort features specifications of loop termination measures at the lines 11 and 17. To state and prove a correctness theorem

[8]For now, we do not implement any clever strategies for efficient constrained input generation for a given precondition.

w.r.t. to given pre-/postconditions for a method, Velvet provides the **`prove_correct`** command (line 22), which takes a method's name and generates the corresponding VCs derived from **`requires`** and **`ensures`** annotations, as well as loop invariants and termination measures (*cf.* Sec. 5). The **`loom_solve`** commands first generates one Lean proof goal per VC, and then runs a customisable automation tactic to discharge each goal. If automation fails on at least one goal, the respective annotation gets highlighted in the editor, and the residual proof obligation is left to the user to prove. For instance, imagine the user forgot to add `n := n + 1` at line 20, so that the measure stated at line 11 does not decrease, and, consequently, an error message with the goal obligation depicted at Fig. 12 will be emitted to prove interactively. While in this case the goal is not provable, often, the residual obligations might be valid but out of reach for the available automation. In this case, nothing prevents the user from attempting to prove them manually using Lean proof mode.

Unlike existing intrinsic verifiers [55, 58, 76], which bundle all proof obligations into a single task, Velvet makes it possible to decouple proofs of partial and total correctness. This can be done by defining two identical version of a program, one without termination-related annotations (highlighted in red in Fig. 11), and another without functional correctness-specific ones (highlighted in yellow). The proof of the former constitutes partial correctness, and the latter proves termination. We can then use the following theorem provided by Velvet to prove the program's total correctness:

```
lemma partial_total_split {α} : ∀ (c₁ c₂ : VelvetM α) (P : Prop) (Q : α → Prop),
 eraseEq c₁ c₂ → triplePartial P c₁ Q → tripleTotal P c₂ (λ _, True) → tripleTotal P c₁ Q
```

8.2 Multiple Verification Modes for Sparse Data Computations

To exercise Velvet's automated/interactive proof capabilities, we implemented several programs that perform multiplications of sparse matrices and vectors. Such computations are known to be challenging to verify mechanically [7, 27, 48]. To the best of our knowledge, only two approaches offer computer-aided techniques to verify computations with multiple sparse structures [35, 52].

Fig. 13 shows a Velvet signature of `SpMSpV` method. It takes a compressed sparse matrix (most of whose elements are zeros) and a sparse vector, and returns the dot-product of decompressed inputs' counterparts. A classical sequential `SpMSpV` algorithm is implemented in the "two-finger merge" style [50], iterating over each compressed row of the matrix and multiplying it by the input sparse vector. To model the natural parallelism in `SpMSpV`, in our implementation, instead of iterating over each row of the matrix sequentially, we use a simple non-deterministic scheduler to randomly interleave those iterations, following a conventional way to encode concurrency in Dafny [60]. To verify it, we follow a so-called *two-layered paradigm* [6], distilling the part of the proof amendable to SMT solvers from mathematical reasoning involving complex properties of $\sum$. Interestingly, this approach has already been studied previously in Rocq, although without automation in mind [48].

```
method SpMSpV (spm: SpM) (spv: SpV)
   return (out: arrVal)
ensures size out = size spm
ensures ∀ i < spm.size, out[i] =
   ∑ j ∈ spv.size, spv[j] * spm[i][j]
```

Fig. 13. SpMSpV method signature

First, we come up with `SpMSpV_pure`: a recursive functional analogue of `SpMSpV` that does all the computations sequentially. `SpMSpV_pure` can be used in the specification, so that we can verify in Velvet that `SpMSpV` returns the same result as `SpMSpV_pure` via SMT, by treating `SpMSpV_pure` as an uninterpreted function with natural definitional equalities. By doing so, we reduce reasoning about effectful `SpMSpV` with loops and mutation to reasoning about pure recursive `SpMSpV_pure`.

Next, we prove that the result of `SpMSpV_pure` is equal to the big summation from Fig. 13 using Lean tactics and facts from the mathlib library [72]. Both in our study and in the prior work [48], the second layer of the proof extensively relied on algebraic properties of $\sum$ and constituted about 300-400 LOC. However, in our case, the first layer of the proof required *no manual reasoning at all.*

9 Related Work

Our work connects several lines of research, most notably: (1) foundational reasoning about effectful computations and (2) interaction between different verification modes in program proofs.

Dijkstra Monads. Dijkstra monads [4, 70, 96] are an established theoretical framework to soundly relate executable semantics of effectful computations to their verification condition generators, and our results are inspired and enabled by the prior work on them. In particular, Ahman et al. proposed a general specification meta-language called DM to define computational monads, so that for a monad M, a monad transformer T, such that $T\ Id = M$, can be derived automatically, and the corresponding specification monad W is defined by applying T to the continuation-passing style (CPS) monad: T CPS [4]. To reason about monadic computations in Ahman et al.'s approach, for each specification monad W, the user has to define a specification of the monadic computation as an element of W, so verification boils down to proving that the specification spec : W of a monadic computation $m : M$ refines its weakest precondition $\mathcal{WP}\ m$. A practical drawback of this approach is that such refinement statements produce VCs that often involve quantification over *all* postconditions. This aspect is undesirable for specifying computations with state *containing functions and relations*, which is the case, *e.g.*, in our implementation of Veil (Sec. 7). Following Ahman et al.'s recipe for it would produce statements with higher-order quantification, not suitable for SMT solvers. In contrast, our work derives generators for Hoare-style VCs for arbitrary specification languages using the definition (8), resulting in more automation-friendly formulas.

Maillard et al. generalise DM to account for a larger class of monads [70]. They also propose a systematic way of deriving Hoare triples: for a monad transformer T, they note that if T is applied to the pre-/postcondition monad PrePost $\triangleq$ Prop $\times$ ($\alpha \longrightarrow$ Prop), then the resulting specification monad T PrePost allows one to derive Hoare-style statements. Unfortunately, the definition of the PrePost monad comes with a subtle issue. To wit, consider its definition of bind given below:

$$\mathsf{bind}^{\mathsf{PrePost}}\ p\ f \triangleq \langle\ (pre \land \forall a,\ post\ a \Rightarrow pre'\,a), \lambda b.\ \exists a, post\ a \land post'\ a\ b\ \rangle$$

where $p = \langle pre, post\rangle$ and $f = \lambda a.\ \langle pre'\ a, post'\ a\rangle$. This definition introduces an existential quantifier for *every* sequential composition in a program. An ad-hoc version of such VC encoding has been studied before and shown to be ill-suited for SMT, unlike the one obtained via $\mathcal{WP}$ [57].

Both approaches [4, 70] have been implemented in the F* dependently-typed language and verifier [95]. Unlike Lean, F* does not allow one to manipulate program semantics as first-class citizens, which means that it cannot be used as a prover to formalise the meta-theory of the respective VC generators, as we did for Loom in Lean. The implementation of Dijkstra monads in F* is, therefore, not foundational (Maillard et al.'s results are mechanised in Rocq but are separate from their F* implementation). Pragmatically, it also means that one cannot state in F* our theorem from Sec. 8.1 that disentangles a proof of a program's partial correctness from its termination proof.

Interaction Trees. Vistrup et al. proposed the *Program Logics à la Carte* (PLC) framework to incrementally derive Separation Logics [85] for programs with algebraic effects [99] by defining their semantics in terms of *interaction trees* (ITrees) [102]. While this approach has been shown to be very expressive, it is not geared towards providing an executable semantics and proof automation. To execute programs whose semantics is defined via ITrees, one has to provide a custom step-by-step termination-ensuring interpreter, eliminating one effect in the tree at a time. Our approach works directly with Lean monads, and allows one to verify and run Lean code using its native executable semantics. In particular, this made it posible to have a verified a randomised simulator for distributed protocols directly in Loom (Sec. 7.2). Such simulator is implemented using Lean's standard Gen monad [56]. As we can run it using native Lean compiler, the obtained checker is very fast, enabling randomised testing of complex distributed protocols, such as NOPaxos. Finally, VCs generated by Loom are designed to be SMT-friendly, while PLC produces VCs in the form of specific Separation Logic formulas with many higher-order constructions. Making such VCs amenable to SMT is non-trivial even for simpler versions of Separation Logic [28, 61, 81].

Unifying verification and symbolic execution. The theoretical foundations of Loom accommodate multiple styles of symbolic reasoning about programs: in addition to Hoare-style correctness proofs, it also allows for reasoning about reachability in the style of Incorrectness Logic [77] (*cf.* Sec. 6.1). In this capacity, our work complements existing efforts on unifying correctness and incorrectness proofs in the presence of different computational effects, such as Outcome Logic (OL) [107, 108] and Hyper Hoare Logic (HHL) [21]. Unlike Loom, none of these logics have been implemented in a form of a foundational program verifier. Similar in spirit to our proofs for a symbolic execution semantics and Veil's testing framework, Correnson and Steinhöfel developed a formally verified symbolic bug finder for a toy WHILE-style language in Rocq [20]. Unlike our work, that effort does not consider combinations of effects or deductive verification.

Combining automated and interactive foundational proofs. RefinedC [90] and RefinedRust [33] are mostly automated foundational verifiers for C and Rust, respectively, based on the Iris program logic [46] and embedded into Rocq. Unlike Loom, they do not provide generic abstractions to define arbitrary effectful semantics and are not optimised for off-the-shelf SMT automation, relying on domain-specific tactics instead. Daenerys [94] is another recent Iris-based verification framework that enhances it with Viper-style SMT-based automation [76] by unifying the semantics of Iris with that of *implicit dynamic frames* [93]. While Daenerys allows for combining automated and interactive verification in Rocq using Iris Proof Mode [53], it does not provide immediately executable semantics and does not support lightweight validation by symbolic execution. It is also unclear whether Daenerys is suitable for embedding domain-specific verifiers, such as Veil.

Multi-modal verifiers. $\mathbb{K}$ [87] is a framework for defining programming language semantics and deriving formal analysis tools, based on *matching logic* [88]. It is expressive and has been used to model the semantics of production languages, including Java [11], ECMAScript 5.1 [79], C11 [29, 39], EVM [41], and Go [105]. The $\mathbb{K}$ framework provides tools for concrete and symbolic execution, model checking, and deductive verification. Unlike Loom, $\mathbb{K}$ is not foundational and has no interactive proof mode. However, matching logic has been formalised in Rocq (with a proof mode not connected to $\mathbb{K}$) [9] and Metamath [18], with its concrete execution and deductive verification backends producing Metamath-checkable certificates for a significant, but incomplete subset of $\mathbb{K}$ features [18, 69]. By contrast, Loom is fully machine-checkable by construction.

Veil is heavily inspired by Ivy [73, 78], a multi-modal verification tool for distributed algorithms. Ivy supports deductive verification backed by SMT solvers, symbolic model checking, and manual proofs using tactics. It also supports extraction of C++ code for execution. Unlike Veil, Ivy is not foundational and does not come with soundness guarantees. Moreover, its support for manual proofs is not interactive, *i.e.*, the user does not get to see a representation of the symbolic context.

10 Conclusion

We have presented Loom—a framework for automatically generating foundational verifiers for executable effectful programs, shallowly embedded into Lean. The machinery of Loom is enabled by a novel theory of *monad transformer algebras*, which we introduced in this work and instantiated for a variety of composable computational effects. With the help of two non-toy verifiers built on top of Loom, Veil (Sec. 7) and Velvet (Sec. 8), we have implemented, tested, and verified correctness of more than 25 case studies, combining SMT-powered automation with interactive Lean proofs.

We believe, our contributions open several avenues for future work. On a theoretical side, we will explore applications of monad transformer algebras to interaction trees [92] and, with the support for coinduction coming to Lean soon, extend Loom for concurrency [32]. On a practical side, we will extend our implementations of Veil and Velvet to further explore interactions between lightweight validation methods and proofs, such as invariant inference [103] and proof repair [36].

Acknowledgments

We thank Shaowei Lin, Peter Müller, and Alex Potanin for their feedback on drafts of this paper. We also thank the anonymous POPL'26 reviewers for their constructive and insightful comments; we are particularly grateful to Reviewer C for their detailed comments on the relation between monad transformer algebras (Sec. 3.4) and the work by Maillard et al. [70]. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001, by Stellar Development Foundation Academic Research Grant, and by Sui Academic Research Award.

References

[1] Smbat Abian and Arthur B. Brown. 1961. A Theorem on Partially Ordered Sets, With Applications to Fixed Point Theorems. *Canadian Journal of Mathematics* 13 (1961), 78–82. https://doi.org/10.4153/CJM-1961-007-5
[2] Jirí Adámek, Stefan Milius, Nathan J. Bowler, and Paul Blain Levy. 2012. Coproducts of Monads on Set. In *LICS*. IEEE Computer Society, 45–54. https://doi.org/10.1109/LICS.2012.16
[3] Reynald Affeldt, Jacques Garrigue, David Nowak, and Takafumi Saikawa. 2021. A trustful monad for axiomatic reasoning with probability and nondeterminism. *Journal of Functional Programming* 31 (2021), e17. https://doi.org/10.1017/S0956796821000137
[4] Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra Monads for Free. (2017), 515–529. https://doi.org/10.1145/3009837.3009878
[5] Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *ESOP (LNCS, Vol. 6602)*. Springer, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1
[6] Andrew W. Appel. 2022. Coq's Vibrant Ecosystem for Verification Engineering (Invited Talk). In *CPP*. ACM, 2–11. https://doi.org/10.1145/3497775.3503951
[7] Gilad Arnold, Johannes Hölzl, Ali Sinan Köksal, Rastislav Bodík, and Mooly Sagiv. 2010. Specifying and verifying sparse matrix codes. In *ICFP*. ACM, 249–260. https://doi.org/10.1145/1863543.1863581
[8] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *TACAS (LNCS, Vol. 13243)*. Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24
[9] Péter Bereczky, Xiaohong Chen, Dániel Horpácsi, Tamás Bálint Mizsei, Lucas Peña, and Jan Tusil. 2022. Mechanizing Matching Logic in Coq. In *Proceedings of the Sixth Working Formal Methods Symposium (FROM) (EPTCS, Vol. 369)*. 17–36. https://doi.org/10.4204/EPTCS.369.2
[10] Yves Bertot and Vladimir Komendantsky. 2008. Fixed point semantics and partial recursion in Coq. In *PPDP*. ACM, 89–96. https://doi.org/10.1145/1389449.1389461
[11] Denis Bogdanas and Grigore Rosu. 2015. K-Java: A Complete Semantics of Java. In *POPL*. ACM, 445–456. https://doi.org/10.1145/2676726.2676982
[12] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *CCS*. ACM, 1032–1043. https://doi.org/10.1145/2976749.2978428
[13] Karnbongkot Boonriong, Stefan Zetzsche, and Alastair F. Donaldson. 2025. Compiler Fuzzing in Continuous Integration: A Case Study on Dafny. In *ICST*. IEEE, 441–452. https://doi.org/10.1109/ICST62969.2025.10988954
[14] Manfred Broy and Martin Wirsing. 1981. On the Algebraic Specification of Nondeterministic Programming Languages. In *CAAP (LNCS, Vol. 112)*. Springer, 162–179. https://doi.org/10.1007/3-540-10828-9_61
[15] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
[16] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90. https://doi.org/10.1145/2408776.2408795

[17] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *SOSP*. ACM, 18–37. https://doi.org/10.1145/2815400.2815402

[18] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Rosu. 2021. Towards a Trustworthy Semantics-Based Language Framework via Proof Generation. In *CAV (LNCS, Vol. 12760)*. Springer, 477–499. https://doi.org/10.1007/978-3-030-81688-9_23

[19] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*. ACM, 268–279. https://doi.org/10.1145/351240.351266

[20] Arthur Correnson and Dominic Steinhöfel. 2023. Engineering a Formally Verified Automated Bug Finder. In *ES-EC/FSE*. ACM, 1165–1176. https://doi.org/10.1145/3611643.3616290

[21] Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1485–1509. https://doi.org/10.1145/3656437

[22] 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 (2025), 569–599. https://doi.org/10.1145/3704856

[23] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[24] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *CADE (LNCS, Vol. 9195)*. Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[25] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

[26] Mike Dodds. 2024. *N* things I learned trying to do formal methods in industry. Available at https://mikedodds.github.io/files/talks/2024-10-09-n-things-I-learned.pdf.

[27] Tristan Dyer, Alper Altuntas, and John W. Baugh Jr. 2019. Bounded Verification of Sparse Matrix Computations. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 36–43. https://doi.org/10.1109/Correctness49594.2019.00010

[28] Marco Eilers, Malte Schwerhoff, and Peter Müller. 2024. Verification Algorithms for Automated Separation Logic Verifiers. In *CAV (LNCS, Vol. 14681)*. Springer, 362–386. https://doi.org/10.1007/978-3-031-65627-9_18

[29] Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. In *POPL*. ACM, 533–544. https://doi.org/10.1145/2103656.2103719

[30] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *IEEE Symposium on Security and Privacy*. IEEE. https://doi.org/10.1109/SP.2019.00005

[31] Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32.

[32] Simon Foster, Chung-Kil Hur, and Jim Woodcock. 2025. Unifying Model Execution and Deductive Verification with Interaction Trees in Isabelle/HOL. *ACM Trans. Softw. Eng. Methodol.* 34, 4 (2025). https://doi.org/10.1145/3702981

[33] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1115–1139. https://doi.org/10.1145/3656422

[34] Martin Giese and Wolfgang Ahrendt. 1999. Hilbert's epsilon-Terms in Automated Theorem Proving. In *TABLEAUX (LNCS, Vol. 1617)*. Springer, 171–185. https://doi.org/10.1007/3-540-48754-9_17

[35] Vladimir Gladshtein, Qiyuan Zhao, Willow Ahrens, Saman P. Amarasinghe, and Ilya Sergey. 2024. Mechanised Hypersafety Proofs about Structured Data. *Proc. ACM Program. Lang.* 8, PLDI (2024), 647–670. https://doi.org/10.1145/3656403

[36] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI (2023), 25–49. https://doi.org/10.1145/3591221

[37] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI*. USENIX Association, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[38] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 716–744. https://doi.org/10.1145/3622823

[39] Chris Hathhorn, Chucky Ellison, and Grigore Rosu. 2015. Defining the undefinedness of C. In *PLDI*. ACM, 336–345. https://doi.org/10.1145/2737924.2737979

[40] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *SOSP*. ACM, 1–17. https://doi.org/

10.1145/2815400.2815428
[41] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *CSF*. IEEE Computer Society, 204–217. https://doi.org/10.1109/CSF.2018.00022
[42] Ralf Hinze. 2000. Deriving backtracking monad transformers. In *ICFP*. ACM, 186–197. https://doi.org/10.1145/351240.351258
[43] Son Ho and Clément Pit-Claudel. 2024. Incremental Proof Development in Dafny with Module-Based Induction. In *Proceedings of the First Workshop on Dafny*.
[44] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP (2022), 711–741. https://doi.org/10.1145/3547647
[45] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259
[46] The Iris Project. 2024. The Iris 4.3 Reference. https://iris-project.org/ Online; last accessed 8 July 2025.
[47] Shin-ya Katsumata. 2014. Parametric effect monads and semantics of effect systems. In *POPL*. ACM, 633–646. https://doi.org/10.1145/2535838.2535846
[48] Ariel E. Kellison, Andrew W. Appel, Mohit Tekriwal, and David Bindel. 2023. LAProof: A Library of Formal Proofs of Accuracy and Correctness for Linear Algebra Programs. In *ARITH*. IEEE, 36–43. https://doi.org/10.1109/ARITH58626.2023.00021
[49] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *ICFP*. ACM, 192–203. https://doi.org/10.1145/1086365.1086390
[50] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 77:1–77:29. https://doi.org/10.1145/3133901
[51] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *SOSP*. ACM, 207–220. https://doi.org/10.1145/1629575.1629596
[52] Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. 2023. Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 154 (2023). https://doi.org/10.1145/3591268
[53] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. ACM, 205–217. https://doi.org/10.1145/3009837.3009855
[54] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *POPL*. ACM, 179–192. https://doi.org/10.1145/2535838.2535841
[55] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *SOSP*. ACM, 438–454. https://doi.org/10.1145/3694715.3695952
[56] leanprover-community. 2025. Plausible: A property testing framework for Lean 4. https://github.com/leanprover-community/plausible. Last accessed on 9 July 2025.
[57] K. Rustan M. Leino. 2005. Efficient weakest preconditions. *Inf. Process. Lett.* 93, 6 (2005), 281–288. https://doi.org/10.1016/J.IPL.2004.10.015
[58] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (LNCS, Vol. 6355)*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20
[59] K. Rustan M. Leino. 2015. Compiling Hilbert's epsilon operator. In *LPAR (EPiC Series in Computing, Vol. 35)*. EasyChair, 106–118. https://doi.org/10.29007/RKXM
[60] K. Rustan M. Leino. 2018. Modeling Concurrency in Dafny. In *Engineering Trustworthy Software Systems*, Jonathan P. Bowen, Zhiming Liu, and Zili Zhang (Eds.). Springer International Publishing, Cham, 115–142.
[61] K. Rustan M. Leino and Peter Müller. 2009. A Basis for Verifying Multi-threaded Programs. In *ESOP (LNCS, Vol. 5502)*. Springer, 378–393. https://doi.org/10.1007/978-3-642-00590-9_27
[62] Xavier Leroy. 2006. Coinductive Big-Step Operational Semantics. In *ESOP (LNCS, Vol. 3924)*. Springer, 54–68. https://doi.org/10.1007/11693024_5
[63] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. ACM, 42–54. https://doi.org/10.1145/1111037.1111042
[64] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. 2021. Modular verification of programs with effects and effects handlers. *Formal Aspects Comput.* 33, 1 (2021), 127–150. https://doi.org/10.1007/S00165-020-00523-2
[65] Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *4th Conference on Computability in Europe (CiE) (LNCS, Vol. 5028)*. Springer, 359–369. https://doi.org/10.1007/978-3-540-69407-6_39

[66] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *OSDI*. USENIX Association, 467–483. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li

[67] Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *POPL*. ACM Press, 333–343. https://doi.org/10.1145/199448.199528

[68] Jannis Limperg and Asta Halkjær From. 2023. Aesop: White-Box Best-First Proof Search for Lean. In *CPP*. ACM, 253–266. https://doi.org/10.1145/3573105.3575671

[69] Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Rosu. 2023. Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 56–84. https://doi.org/10.1145/3586029

[70] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra monads for all. *Proc. ACM Program. Lang.* 3, ICFP (2019), 104:1–104:29. https://doi.org/10.1145/3341708

[71] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *ESOP (LNCS, Vol. 11423)*. Springer, 30–59. https://doi.org/10.1007/978-3-030-17184-1_2

[72] The mathlib Community. 2020. The Lean mathematical library. In *CPP*. ACM, 367–381. https://doi.org/10.1145/3372885.3373824 https://github.com/leanprover-community/mathlib4.

[73] Kenneth L. McMillan and Oded Padon. 2020. Ivy: A Multi-modal Verification Tool for Distributed Algorithms. In *CAV (LNCS, Vol. 12225)*. Springer, 190–202. https://doi.org/10.1007/978-3-030-53291-8_12

[74] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *CAV (LNCS, Vol. 8044)*. Springer, 696–701. https://doi.org/10.1007/978-3-642-39799-8_48

[75] Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark Barrett. 2025. Lean-SMT: An SMT tactic for discharging proof goals in Lean. In *CAV*. To appear.

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

[77] Peter W. O'Hearn. 2020. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 10:1–10:32. https://doi.org/10.1145/3371078

[78] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *PLDI*. ACM, 614–630. https://doi.org/10.1145/2908080.2908118

[79] Daejun Park, Andrei Stefanescu, and Grigore Rosu. 2015. KJS: a complete formal semantics of JavaScript. In *PLDI*. ACM, 346–356. https://doi.org/10.1145/2737924.2737991

[80] Arthur Paulino, Damiano Testa, Edward Ayers, Evgenia Karunus, Henrik Böving, Jannis Limperg, Siddhartha Gadgil, and Siddharth Bhat. 2024. Metaprogramming in Lean 4. Available at https://leanprover-community.github.io/lean4-metaprogramming-book/.

[81] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *CAV (LNCS, Vol. 8044)*. Springer, 773–789. https://doi.org/10.1007/978-3-642-39799-8_54

[82] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *IEEE Symposium on Security and Privacy*. IEEE, 983–1002. https://doi.org/10.1109/SP40000.2020.00114

[83] George Pîrlea, Vladimir Gladshtein, Elad Kinsbruner, Qiyuan Zhao, and Ilya Sergey. 2025. Veil: A Framework for Automated and Interactive Verification of Transition Systems. In *CAV*. Springer. To appear.

[84] Yicheng Qian, Joshua Clune, Clark Barrett, and Jeremy Avigad. 2025. Lean-auto: An Interface between Lean 4 and Automated Theorem Provers. In *CAV*. To appear.

[85] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[86] Rocq Development Team. 2025. The Rocq Prover. https://rocq-prover.org. Version 9.0.0, released March 12, 2025.

[87] Grigore Rosu. 2017. $\mathbb{K}$: A Semantic Framework for Programming Languages and Formal Analysis Tools. In *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 50. IOS Press, 186–206. https://doi.org/10.3233/978-1-61499-810-5-186

[88] Grigore Rosu. 2017. Matching Logic. *Log. Methods Comput. Sci.* 13, 4 (2017). https://doi.org/10.23638/LMCS-13(4:28)2017

[89] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.* 79, 6 (2010), 397–434. https://doi.org/10.1016/J.JLAP.2010.03.012

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

[91] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized Verification of Fine-Grained Concurrent Programs. In *PLDI*. ACM, 77–87. https://doi.org/10.1145/2737924.2737964

[92] Lucas Silver and Steve Zdancewic. 2021. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434307

[93] Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP (LNCS, Vol. 5653)*. Springer, 148–172. https://doi.org/10.1007/978-3-642-03013-0_8

[94] 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). https://doi.org/10.1145/3729284

[95] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *ICFP*. ACM, 266–278. https://doi.org/10.1145/2034773.2034811

[96] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-Order Programs with the Dijkstra Monad. In *PLDI*. ACM, 387–398. https://doi.org/10.1145/2491956.2491978

[97] Sebastian Ullrich and Leonardo de Moura. 2022. 'do' unchained: embracing local imperativity in a purely functional language (functional pearl). *Proc. ACM Program. Lang.* 6, ICFP (2022), 512–539. https://doi.org/10.1145/3547640

[98] Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover*. Ph. D. Dissertation. University of California, San Diego, USA. http://www.escholarship.org/uc/item/8dm057ws

[99] Max Vistrup, Michael Sammler, and Ralf Jung. 2025. Program Logics à la Carte. *Proc. ACM Program. Lang.* 9, POPL (2025), 300–331. https://doi.org/10.1145/3704847

[100] James R. Wilcox, Yotam M. Y. Feldman, Oded Padon, and Sharon Shoham. 2024. mypyvy: A Research Platform for Verification of Transition Systems in First-Order Logic. In *CAV (LNCS, Vol. 14682)*. Springer, 71–85. https://doi.org/10.1007/978-3-031-65630-9_4

[101] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*. ACM, 357–368. https://doi.org/10.1145/2737924.2737958

[102] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

[103] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *OSDI*. USENIX Association, 485–501. https://www.usenix.org/conference/osdi22/presentation/yao

[104] Mark Yuen. 2022. *Verifying Distributed Protocols: from Executable to Decidable*. Capstone Thesis. Yale-NUS College, Singapore. Accompanying code available at https://github.com/markyuen/tlaplus-to-ivy/.

[105] Can Zhao, Qin Liu, Zonghua Hu, Ze Yu, Dejun Wang, and Bo Meng. 2023. K-Go: An executable formal semantics of Go language in K framework. *IET Blockchain* 3, 2 (2023), 61–73. https://doi.org/10.1049/BLC2.12024

[106] Qiyuan Zhao, George Pîrlea, Karolina Grzeszkiewicz, Seth Gilbert, and Ilya Sergey. 2024. Compositional Verification of Composite Byzantine Protocols. In *CCS*. ACM, 34–48. https://doi.org/10.1145/3658644.3690355

[107] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 522–550. https://doi.org/10.1145/3586045

[108] Noam Zilberstein, Angelina Saliling, and Alexandra Silva. 2024. Outcome Separation Logic: Local Reasoning for Correctness and Incorrectness with Computational Effects. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 276–304. https://doi.org/10.1145/3649821

[109] Maaike Zwart and Dan Marsden. 2022. No-Go Theorems for Distributive Laws. *Logical Methods in Computer Science* Volume 18, Issue 1 (Jan. 2022). https://doi.org/10.46298/lmcs-18(1:13)2022