The Future is Ours: Prophecy Variables in Separation Logic

RALF JUNG, MPI-SWS, Germany
RODOLPHE LEPIGRE, MPI-SWS, Germany
GAURAV PARTHASARATHY, ETH Zurich, Switzerland and MPI-SWS, Germany
MARIANNA RAPOPORT, University of Waterloo, Canada and MPI-SWS, Germany
AMIN TIMANY, imec-DistriNet, KU Leuven, Belgium
DEREK DREYER, MPI-SWS, Germany
BART JACOBS, imec-DistriNet, KU Leuven, Belgium

Early in the development of Hoare logic, Owicki and Gries introduced *auxiliary variables* as a way of encoding information about the *history* of a program's execution that is useful for verifying its correctness. Over a decade later, Abadi and Lamport observed that it is sometimes also necessary to know in advance what a program will do in the *future*. To address this need, they proposed *prophecy variables*, originally as a proof technique for refinement mappings between state machines. However, despite the fact that prophecy variables are a clearly useful reasoning mechanism, there is (surprisingly) almost no work that attempts to integrate them into Hoare logic. In this paper, we present the first account of prophecy variables in a Hoare-style program logic that is flexible enough to verify *logical atomicity* (a relative of linearizability) for classic examples from the concurrency literature like RDCSS and the Herlihy-Wing queue. Our account is formalized in the Iris framework for separation logic in Coq. It makes essential use of *ownership* to encode the exclusive right to resolve a prophecy, which in turn lets us enforce soundness of prophecies with a very simple set of proof rules.

CCS Concepts: • **Theory of computation → Separation logic**; *Programming logic*; Operational semantics.

Additional Key Words and Phrases: Prophecy variables, separation logic, logical atomicity, linearizability, Iris

ACM Reference Format:
Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The Future is Ours: Prophecy Variables in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 45 (January 2020), 32 pages. https://doi.org/10.1145/3371113

1 INTRODUCTION

When proving correctness of a program P, it is often easier and more natural to reason *forward*—that is, to start at the beginning of P's execution and reason about how it behaves as it executes. But sometimes strictly forward reasoning is not good enough: when reasoning about a program step s_0, it may be necessary to "peek into the future" and know ahead of time what will happen at some future program step s_1.

Authors' addresses: Ralf Jung, MPI-SWS, Saarland Informatics Campus, Germany, jung@mpi-sws.org; Rodolphe Lepigre, MPI-SWS, Saarland Informatics Campus, Germany, lepigre@mpi-sws.org; Gaurav Parthasarathy, Department of Computer Science, ETH Zurich, Switzerland and MPI-SWS, Germany, gaurav.parthasarathy@inf.ethz.ch; Marianna Rapoport, University of Waterloo, Canada and MPI-SWS, Germany, mrapoport@uwaterloo.ca; Amin Timany, imec-DistriNet, KU Leuven, Belgium, amin.timany@cs.kuleuven.be; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org; Bart Jacobs, imec-DistriNet, KU Leuven, Belgium, bart.jacobs@cs.kuleuven.be.

2475-1421/2020/1-ART45
https://doi.org/10.1145/3371113

To address this need, Abadi and Lamport [1988, 1991] introduced the idea of *prophecy variables*. Prophecy variables are a form of *auxiliary variable*—a "logical" or "ghost" variable that encodes state information relevant to the proof of a program that is not present in the physical state of the program itself. Auxiliary variables were originally proposed by Owicki and Gries [1976] in the form of *history variables*, which record information about what has happened in an execution so far (*the past*). In contrast, prophecy variables supply information about what will happen later on in the execution (*the future*). The focus of Abadi and Lamport's original paper was on using both history and prophecy variables to prove that one program (or program specification) is a correct implementation of another, by showing that the first *refines*—*i.e.*, has a subset of the observable behaviors of—the second. Their main result was a theorem establishing that, under some restrictions, the combination of history and prophecy variables offers a sound and complete technique for proving valid refinement mappings.

However, despite the duality and complementarity of history and prophecy variables, there is a striking difference between the formal settings in which these mechanisms have been deployed:

- History variables were introduced in the 1970s in the context of Hoare logic. They—along with their modern descendants like *user-defined ghost state* [Dinsdale-Young et al. 2013; Ley-Wild and Nanevski 2013; Turon et al. 2014; Jung et al. 2015; Sergey et al. 2015; Jung et al. 2018]—continue to this day to play an important role in deductive program verification.
- Prophecy variables were introduced as a tool for establishing refinement mappings between state machines, but compared to history variables they remain a fairly exotic and under-explored technique. Moreover, although prophecy variables have been integrated into verification tools based on reduction [Sezgin et al. 2010] and temporal logic [Cook and Koskinen 2011; Lamport and Merz 2017], there has been almost no work at all on incorporating prophecy variables into a Hoare-style program logic.

1.1 Prior Work on Using Prophecy Variables in Hoare Logic

To our knowledge, there are only two pieces of prior work that utilize prophecy variables in the context of Hoare logic. However, one of them suffers from a serious, previously undiscovered technical flaw, and the other is quite limited in its expressiveness.

The first is Vafeiadis's PhD thesis [Vafeiadis 2008], in which he shows how to prove *linearizability*—a standard correctness criterion for concurrent data structures [Herlihy and Wing 1990]—using RGSep [Vafeiadis and Parkinson 2007], a modern variety of *separation logic* [Reynolds 2002]. To prove linearizability using Vafeiadis's method involves finding a "linearization point" in each concurrent operation—*i.e.*, an instant during the execution of the operation when the operation appears to atomically take effect—and updating a relevant bit of "ghost state" at that point. For some more advanced data structures, though, the location of the linearization point may not be evident at the point in time when it occurs, but perhaps only later, at a future step of execution. Vafeiadis gives one such example, the RDCSS data structure [Harris et al. 2002], and uses prophecy variables to guess the location of the linearization point ahead of time. But the treatment of prophecy variables in his proof is informal: he suggests that they are a sound technique by reference to Abadi and Lamport's paper, but does not formalize their integration into his program logic.

The second piece of prior work is due to Zhang et al. [2012], who present a "structural" approach to prophecy variables, motivated explicitly by the desire to put Vafeiadis's technique on a more formal footing. What makes their approach "structural" is that, to simplify the proof of soundness, their prophecy variables have restricted scope: at program step s_0, one can guess the result of what will happen at future step s_1 only if both s_0 and s_1 occur in the same syntactic block. In particular, their approach does not allow one *thread* to prophesy the result of a future step executed by a

different thread. Nevertheless, they claim that it is sufficient to handle the RDCSS example from Vafeiadis's thesis because his use of prophecy variables follows this structural discipline.

Unfortunately, we have discovered that Vafeiadis's proof of RDCSS is flawed. (We reported the flaw to him, and he has confirmed it.) Even more unfortunately, the flaw in Vafeiadis's proof pertains directly to his use of prophecy variables that follow the structural discipline. As we argue in §6.1, for verifying RDCSS it seems necessary to employ non-structural prophecies (*e.g.*, prophecies across threads).

1.2 Our Contribution: Accounting for Prophecy Variables in Separation Logic

In this paper, we present the first account of prophecy variables in a Hoare-style program logic that is sufficiently flexible to verify classic examples from the concurrency literature, such as RDCSS [Harris et al. 2002] and the Herlihy-Wing queue [Herlihy and Wing 1990]. Unlike the original work on prophecies, we make no claims about completeness, and we focus solely on safety properties. Rather, our goal is to establish prophecy variables as a viable and useful addition to the Hoare logic toolbox.

The key idea behind our approach is to model prophecy variables as an *ownable resource* in the context of *concurrent separation logic* [O'Hearn 2007]. When we create a prophecy variable p in a separation-logic proof, we will obtain exclusive ownership of a *prophecy assertion* $\mathsf{Proph}(p, v)$ for some value v. This assertion tells us two things: (1) "the future is ours"—we own the *exclusive right to resolve* (*i.e.*, assign a value to) p—and yet (2) "we cannot escape our destiny"—p is prophesied to be resolved to v, meaning that if p does get resolved in the future of the program's execution, then it will be resolved to v. In order to verify a subsequent step of execution in which p is resolved to a value w, we must give up ownership of $\mathsf{Proph}(p, v)$, but in return we learn that $v = w$ (*i.e.*, that our prophecy was correct). The exclusivity of $\mathsf{Proph}(p, v)$ here serves to guarantee that p can be resolved at most once, so we do not have to worry about it being resolved in inconsistent ways by different threads. Thus, by buying into the framework of concurrent separation logic, we get soundness of prophecies for free!

Furthermore, we are not restricted to one-shot prophecies: we also support a more general form of prophecy variables that can get resolved multiple times.[1] When such a variable p is created, it will prophesy the *sequence* of values with which it will be resolved (*i.e.*, $\mathsf{Proph}(p, [v_1, \ldots, v_n])$). We can then use this exclusive assertion to resolve p multiple times, threading ownership of it through the resolutions but popping another value v_i off the head of the sequence each time we resolve. Such sequence prophecies are useful, for example, when proving linearizability of a concurrent data structure, because they allow us to predict the ordering of concurrent operations in advance.

We develop our formal account of prophecy variables in the higher-order concurrent separation logic framework *Iris* [Jung et al. 2018]. We have several reasons for doing so. First, Iris provides a modular foundation for rapid prototyping of new separation logics: much of the hard work of proving soundness of a modern separation logic is already handled in the soundness proof of the Iris "base logic", and so the encoding of new logics can be done at a relatively high level of abstraction. Second, Iris is implemented in the Coq proof assistant and provides good tactical support for interactive development of machine-checked separation-logic proofs [Krebbers et al. 2017, 2018].

Last but not least, by developing prophecy variables in Iris, we can at last overcome one of the major limitations of prior work on concurrent separation logic (Iris included). Specifically, several advanced separation logics have established proof techniques for a very strong correctness property for concurrent data structures called *logical atomicity* (a.k.a. "abstract atomicity") [Jacobs

[1] Lamport and Merz [2017] also support multiple resolutions in TLA+ with array and data structure prophecy variables.

```
new_EA_coin() ≜ {val = ref(nondet_bool())}        read_EA_coin(c) ≜ ! c.val
```

Fig. 1. An extremely simple implementation of eager coins.

```
new_LZ_coin() ≜
  let v = ref(None);
  let p = NewProph;
  {val = v, p = p}

read_LZ_coin(c) ≜
  match ! c.val with
    Some(b) ⇒ b
  | None    ⇒ let r = nondet_bool();
              c.val ← Some(r);
              Resolve c.p to r; r
  end
```

Fig. 2. The lazy coin implementation.

and Piessens 2011; da Rocha Pinto et al. 2014; Jung et al. 2015; Frumin et al. 2018]. Logical atomicity can be seen as an "internalization" of linearizability within separation logic: operations on a data structure that are proven to be logically atomic may be reasoned about by clients of the data structure using much stronger proof rules that are normally reserved for physically atomic operations (see §4 for details). However, there are certain concurrent data structures for which it was seemingly impossible to prove logical atomicity in existing separation logics: namely, those data structures (like RDCSS and the Herlihy-Wing queue) whose linearization points can only be determined at some future step of execution. Using prophecy variables, we can finally prove logical atomicity for such data structures in Iris.

The rest of the paper is structured as follows. In §2, we present our separation-logic account of prophecy variables, and show how to use it to verify some simple motivating examples. In §3, we describe how we establish the soundness of prophecy variables in Iris. In §4, using the RDCSS data structure as a motivating example, we review the idea of logical atomicity and how it is proven in Iris. In §5, we explore the implementation of RDCSS in detail and give an intuitive argument for its correctness. In §6, we show how prophecies are useful in proving RDCSS formally in Iris. Finally, in §7, we conclude with related and future work.

All the results in this paper are verified in Coq [Jung et al. 2019]. Our Coq formalization also includes prophecy-based proofs of logical atomicity for several other examples, including an "atomic snapshot" data structure [Sergey et al. 2015] and the Herlihy-Wing queue [Herlihy and Wing 1990]. In addition, we have extended the VeriFast program verifier [Jacobs et al. 2011] with support for prophecy variables based on our separation-logic specification, and used that extension to verify several interesting examples [Jung et al. 2019]. For space reasons, we focus here solely on our account of prophecy variables in Iris.

2 KEY IDEAS

In this section we use a simple motivating example to illustrate the key ideas of how we incorporate prophecy variables into separation logic and use them for reasoning about non-deterministic behavior of programs. We begin by presenting our example (§2.1) to motivate the introduction of *one-shot prophecy variables* (§2.2), which prophesy a single future value. We then present the more general *sequence prophecy variables* (§2.3), which prophesy a sequence of values, and again give an example to illustrate their use. We also show how one-shot prophecies can be derived using sequence prophecies. We conclude the section by introducing *atomic resolutions* of prophecies (§2.4), which are useful when verifying concurrent algorithms.

2.1 Motivating Example: A Specification for Eager and Lazy Coins

Lazy coins. Consider the code in Fig. 1, which presents an extremely simple implementation of *eager coins*. The function `new_EA_coin` flips a coin by non-deterministically choosing some boolean value, and then stores the result in a freshly allocated ref cell. The function `read_EA_coin` simply reads the value of the coin by dereferencing the ref cell. We call these coins "eager" because their value is determined immediately upon their creation. (Note that the use of mutable state here is gratuitous: it merely serves to build a better bridge to the next example.)

One possible Hoare-style specification for these functions is the following:

$$\{\mathsf{True}\}\ \texttt{new\_EA\_coin}()\ \{c.\ \exists b.\ \mathsf{EagerCoin}(c, b)\}$$

$$\{\mathsf{EagerCoin}(c, b)\}\ \texttt{read\_EA\_coin}(c)\ \{v.\ v = b * \mathsf{EagerCoin}(c, b)\}$$

Here, the abstract predicate $\mathsf{EagerCoin}(c, b)$ asserts that the coin c has value b. Thus, when we create a new coin c, we learn that there *exists* some boolean b such that c has the value b; and when we subsequently read c, the result must be b. Note that the postconditions of the two functions contain a binder, which binds the result value of the expression being verified. (This is standard in many modern presentations of separation logic for functional languages. We will omit the binder when the result value is unit, *i.e.*, ().)

To prove this spec, the abstract predicate $\mathsf{EagerCoin}(c, b)$ can be defined simply as follows:

$$\mathsf{EagerCoin}(c, b) \triangleq c.\mathit{val} \mapsto b$$

This is the famous "points-to" predicate of separation logic, which asserts both *ownership* of the ref cell $c.\mathit{val}$ and the *knowledge* that $c.\mathit{val}$ points to b. With this definition in hand, the coin spec follows directly from the basic rules of standard separation logic.

Now consider the implementation of *lazy coins* shown in Fig. 2. For the time being, ignore the code colored in red—it is prophecy-related "ghost code", whose purpose we will explain in a moment.

Lazy coins differ from eager coins in that the coin flip does not take place when a coin is created but rather when the coin is first read. To achieve this, the lazy coin's value is represented as a reference to a boolean *option*. When the coin is created, it starts out with value `None`. Then, when the coin is read for the first time, a boolean b is non-deterministically chosen and the coin's value is updated to `Some` b. Thereafter, all reads of the coin return b.

From the point of view of a client, lazy coins are indistinguishable from eager coins, since there is no way to observe the value of a coin in between its creation and the first time it is read. Therefore, we *ought* to be able to prove the same specification for lazy coins as we did for eager coins:

$$\{\mathsf{True}\}\ \texttt{new\_LZ\_coin}()\ \{c.\ \exists b.\ \mathsf{LazyCoin}(c, b)\}$$

$$\{\mathsf{LazyCoin}(c, b)\}\ \texttt{read\_LZ\_coin}(c)\ \{v.\ v = b * \mathsf{LazyCoin}(c, b)\}$$

However, under the usual semantics for Hoare triples in separation logic, there is no way of defining LazyCoin that would validate this spec. The problem is that, when verifying `new_LZ_coin`(), we don't know how to instantiate the existential quantifier for the boolean value b in the postcondition. The identity of b will only be known later, when c is read for the first time.

This is precisely the type of verification problem that prophecy variables were born to solve.

2.2 One-Shot Prophecies

We begin here by introducing a simple form of prophecy variables we call *one-shot prophecies*. To make use of these prophecies in our verification, we must instrument our code with prophecy *ghost variables*, along with ghost operations for manipulating them. This *ghost code* does not in any way

alter the behavior of the program and is only inserted to facilitate program verification. We use the color red to distinguish ghost code from regular code, and in §3.5, we will formally prove that such ghost code is safely erasable.

The specifications for the one-shot prophecy operations **NewProph** and **Resolve** are as follows:

ONE-SHOT-PROPHECY-CREATION
$\{\mathsf{True}\}\ \mathbf{NewProph}\ \{p.\ \exists v.\ \mathsf{Proph}_1(p, v)\}$

ONE-SHOT-PROPHECY-RESOLUTION
$\{\mathsf{Proph}_1(p, v)\}\ \mathbf{Resolve}\ p\ \mathbf{to}\ w\ \{v = w\}$

The prophecy creation operation **NewProph** returns a fresh prophecy identifier p, along with a *prophecy assertion* $\mathsf{Proph}_1(p, v)$ for *some* existentially-quantified value v. This assertion describes both the exclusive right to resolve the prophecy p, as well as the knowledge that p *will* be resolved to v later in the execution of the program. Given this prophecy assertion, the resolve operation, **Resolve** p **to** w, resolves p to the value w. This resolution guarantees in the postcondition that the prophesied value v is in fact the same as the value w to which p was actually resolved. Crucially, prophecy resolution *consumes* the right to resolve the prophecy: this is essential for ensuring that a one-shot prophecy variable is not resolved more than once in a single execution trace (as that would lead us to a logical inconsistency).

Note that **Resolve** p **to** w places *no restriction* on the value w to which p is resolved. The reader may wonder: if we own $\mathsf{Proph}_1(p, v)$ and we resolve it to $w \neq v$, won't this lead us to a contradiction in our proof? Yes, it will, and that is the whole point! Ownership of $\mathsf{Proph}_1(p, v)$ tells us something about the future execution trace of the program we are currently verifying, namely that p *will* get resolved to v. If we then get to the point of resolving p to a different value, we know we must be in an impossible case of the proof. Indeed, we will see an example of this kind of reasoning in §6.3.

That said, there are situations where it is useful to place some restrictions on the values to which a prophecy is resolved, in the interest of simplifying the proof. For that purpose we introduce *typed* prophecy assertions. Specifically, let V be a nonempty set of values representing a "type". We can define the V-typed prophecy assertion $\mathsf{Proph}_1^V(p, v)$ as follows:

$$\mathsf{Proph}_1^V(p, v) \triangleq \exists v_0.\ \mathsf{Proph}_1(p, v_0) * (v_0 = v \lor v_0 \notin V) \tag{1}$$

We can then derive the following spec for typed one-shot prophecies from the untyped spec above:

ONE-SHOT-PROPHECY-CREATION-TYPED
$\{\mathsf{True}\}\ \mathbf{NewProph}\ \{p.\ \exists v \in V.\ \mathsf{Proph}_1^V(p, v)\}$

ONE-SHOT-PROPHECY-RESOLUTION-TYPED
$\{\mathsf{Proph}_1^V(p, v) * w \in V\}\ \mathbf{Resolve}\ p\ \mathbf{to}\ w\ \{v = w\}$

These are the same as the rules for untyped prophecies, except that: (1) the typed prophecy creation rule is stronger—it *ensures* that p will be resolved to an inhabitant of V—and (2) the typed prophecy resolution rule is weaker—it *requires* that we resolve p to an inhabitant of V. In the coin example, we will make use of *Boolean* prophecies, where V is chosen to be $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$.

The key idea for the proof of ONE-SHOT-PROPHECY-CREATION-TYPED is, in case the actually prophesied value v_0 is *not* in V, to just use any "fake value" $v \in V$. This corresponds to the right disjunct in (1). In ONE-SHOT-PROPHECY-RESOLUTION-TYPED, that case leads to a contradiction since $v_0 = w \in V$.

Verifying the lazy coin spec. With one-shot prophecies in hand, we will now be able to verify the spec for lazy coins shown in §2.1.

The first step is to instrument the implementation of lazy coins with a one-shot prophecy, which will tell the creator of a lazy coin what value the coin will eventually take on. Specifically, we now equip coins with an additional prophecy field p, along with ghost code for manipulating p. When a coin is created, the prophecy p is created along with it. When a coin is read for the first time, the value r gets chosen for it, and the coin's prophecy field p gets resolved to r.

The second step is to define the predicate $\mathsf{LazyCoin}(c, b)$ used in the lazy coin spec. Intuitively, $\mathsf{LazyCoin}(c, b)$ holds either if (1) c has been read already and it stores b, or if (2) c's prophecy field

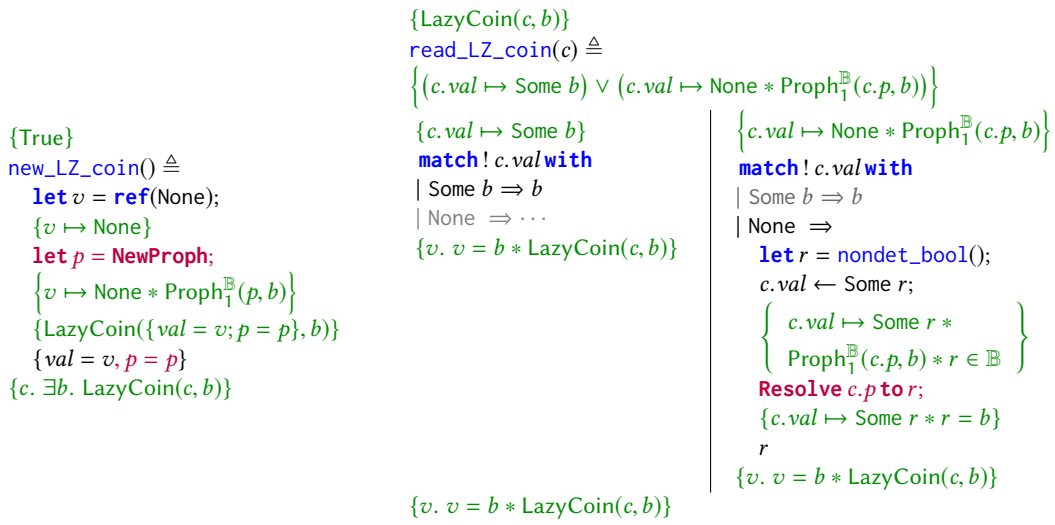


Fig. 3. Proof outline for lazy coin operations.

indicates that it *will* be set to b in the future. Formally, it is defined as follows:

$$\mathsf{LazyCoin}(c, b) \triangleq (c.\mathit{val} \mapsto \mathsf{Some}\ b) \vee (c.\mathit{val} \mapsto \mathsf{None} * \mathsf{Proph}_1^{\mathbb{B}}(c.p, b))$$

Here, the logical connective $*$ is the separating conjunction of separation logic; the proposition $P * Q$ asserts ownership of disjoint resources, one satisfying P and the other Q. Given this definition for LazyCoin, Fig. 3 depicts a proof outline for the lazy coin spec.

Concerning the creation operation: When a lazy coin is created, the value of the coin is None. However, upon creating the prophecy p, we obtain $\mathsf{Proph}_1^{\mathbb{B}}(p, b)$ for some Boolean value b. Hence, we can establish the required postcondition by proving the right disjunct of $\mathsf{LazyCoin}(c, b)$.

Concerning the read operation: There are two different cases to be considered, corresponding to the two disjuncts of LazyCoin; these are shown on either side of the vertical separating bar. The left disjunct of LazyCoin corresponds to the case where the coin already has a value, *i.e.*, the read operation has been called before. This case is straightforward as the stored value is simply returned. In the other case, *i.e.*, on the first call to the read operation, the value of the coin is None. Hence, a Boolean value r is generated non-deterministically and the value Some r is stored in the coin's internal reference. After this, the prophecy is resolved to r. At this point, we obtain that the generated value r equals the prophesied value b, which allows us to recover $\mathsf{LazyCoin}(c, b)$, this time by proving the left disjunct.

2.3 Sequence Prophecies

In more complex scenarios, it is useful to be able to prophesy not only a single future event, but a sequence of future events. For this purpose we introduce *sequence prophecies*.

As motivation for sequence prophecies, consider the implementation of the *clairvoyant coin* shown in Fig. 4, for the moment ignoring the ghost code. The implementation itself is mostly straightforward. The creation operation initializes the coin to a non-deterministically chosen Boolean value, the read operation reads it—and unlike the coins we have considered so far, there is also a *toss* operation that assigns a new, non-deterministically chosen Boolean value to the coin.

```
new_CL_coin() ≜
  let v = ref(nondet_bool());
  let p = NewProph;
  {val = v, p = p}

read_CL_coin(c) ≜
  ! c.val

toss_CL_coin(c) ≜
  let r = nondet_bool();
  c.val ← r;
  Resolve c.p to r
```

Fig. 4. The clairvoyant coin implementation.

$$
\begin{array}{l}
\{\mathsf{ClairvoyantCoin}(c, bs)\} \\
\mathtt{toss\_CL\_coin}(c) \triangleq \\
\left\{bs = b :: bs' * c.val \mapsto b * \mathsf{Proph}^{\mathbb{B}}(c.p, bs')\right\} \\
\quad \mathtt{let}\ r = \mathtt{nondet\_bool}(); \\
\quad c.val \leftarrow r \\
\quad \left\{bs = b :: bs' * c.val \mapsto r * \mathsf{Proph}^{\mathbb{B}}(c.p, bs') * r \in \mathbb{B}\right\} \\
\quad \mathtt{Resolve}\ c.p\ \mathtt{to}\ r; \\
\quad \left\{bs = b :: bs' * bs' = r :: bs'' * c.val \mapsto r * \mathsf{Proph}^{\mathbb{B}}(c.p, bs'')\right\} \\
\{\exists b, bs'.\ bs = b :: bs' * \mathsf{ClairvoyantCoin}(c, bs')\}
\end{array}
$$

Fig. 5. Proof outline for the toss operation of the clairvoyant coin.

What is interesting is the *specification* that we aim to give to this coin implementation, which explains the sense in which it is clairvoyant:

$$\{\mathsf{True}\}\ \mathtt{new\_CL\_coin}()\ \{c.\ \exists bs.\ \mathsf{ClairvoyantCoin}(c, bs)\}$$

$$\{\mathsf{ClairvoyantCoin}(c, bs)\}\ \mathtt{read\_CL\_coin}(c)\ \{x.\ \exists bs'.\ bs = x :: bs' * \mathsf{ClairvoyantCoin}(c, bs)\}$$

$$\{\mathsf{ClairvoyantCoin}(c, bs)\}\ \mathtt{toss\_CL\_coin}(c)\ \{\exists x, bs'.\ bs = x :: bs' * \mathsf{ClairvoyantCoin}(c, bs')\}$$

Here, the predicate $\mathsf{ClairvoyantCoin}(c, bs)$ indicates that bs is the sequence of values that c will take on in the course of the program's execution, with the head of bs being c's current value. Consequently, when we read the coin, we know that the value x we read must be whatever value is at the head of bs; and when we toss the coin, we pop that head element of bs off. Of course, the challenge is that in order to verify the spec for `new_CL_coin`, we must somehow be able to predict the entire sequence of coin flips up front.

This is precisely the functionality that *sequence prophecies* provide. We write $\mathsf{Proph}(p, vs)$ to express that the sequence prophecy variable p prophesies the sequence of values described by vs. Here, vs is in fact a list of pairs of values—and the reason for that will be explained when we introduce atomic resolutions in §2.4—but for the moment we will essentially work with pairs of the form $((), w)$ where $()$ is the element of the unit type and w is a value to which p gets resolved.

The formal specifications for sequence prophecy creation and resolution are as follows:

SEQUENCE-PROPHECY-CREATION
$$\{\mathsf{True}\}\ \mathtt{NewProph}\ \{p.\ \exists vs.\ \mathsf{Proph}(p, vs)\}$$

SEQUENCE-PROPHECY-SIMPLE-RESOLUTION
$$\{\mathsf{Proph}(p, vs)\}\ \mathtt{Resolve}\ p\ \mathtt{to}\ w\ \{\exists vs'.\ vs = ((), w) :: vs' * \mathsf{Proph}(p, vs')\}$$

When p is created using `NewProph`, we obtain $\mathsf{Proph}(p, vs)$ for some sequence vs prophesying all the subsequent resolutions of p. When p gets resolved using `Resolve` p `to` w, the postcondition of the prophecy resolution tells us that $vs = ((), w) :: vs'$ for some sequence vs'. In other words, we

learn that our prophecy was correct: w was the first value to which p was resolved. Furthermore, after resolution, we are left owning $\mathsf{Proph}(p, vs')$, *i.e.*, we have popped the first element w off the sequence, but the remaining predictions remain to be observed (at subsequent resolutions).

Similar to one-shot prophecies, it is again useful to develop a *typed* variant of sequence prophecies. We thus define a predicate $\mathsf{Proph}^V(p, vs)$, similar to $\mathsf{Proph}_1^V(p, v)$ from §2.2, that allows us to enforce that the values we prophesy are drawn from a non-empty set of values V representing a "type" (using notation $(v_1, v_2).2 = v_2$):

$$\mathsf{Proph}^V(p, vs) \triangleq \exists vs'.\ \mathsf{Proph}(p, vs') * |vs| = |vs'| * \forall i < |vs|.\ vs'_i.2 = vs_i \lor vs'_i.2 \notin V$$

Using the untyped rules, we can easily derive the following rules for typed prophecies, where the notation V^* denotes the set of lists of elements of V:

SEQUENCE-PROPHECY-CREATION-TYPED
$$\{\mathsf{True}\}\ \texttt{NewProph}\ \{p.\ \exists vs \in V^*.\ \mathsf{Proph}^V(p, vs)\}$$

SEQUENCE-PROPHECY-SIMPLE-RESOLUTION-TYPED
$$\{\mathsf{Proph}^V(p, vs) * w \in V\}\ \texttt{Resolve}\ p\ \texttt{to}\ w\ \{\exists vs'.\ vs = w :: vs' * \mathsf{Proph}^V(p, vs')\}$$

Going back to the clairvoyant coin example, we can now use typed sequence prophecies to define the ClairvoyantCoin predicate as follows:

$$\mathsf{ClairvoyantCoin}(c, bs) \triangleq \exists b, bs'.\ bs = b :: bs' * c.val \mapsto b * \mathsf{Proph}^{\mathbb{B}}(c.p, bs')$$

The proof of correctness of the toss operation is given in Fig. 5. (We omit the proofs of correctness of the creation and reading of clairvoyant coins since they are rather straightforward.)

Encoding one-shot prophecies with sequences. Unsurprisingly, sequence prophecy variables are more powerful than their one-shot counterpart. In fact, one-shot prophecies can be encoded using sequence prophecies. The idea is to encode the value v of the one-shot prophecy variable as the first predicted resolution of the underlying prophecy sequence. Formally, we define:

$$\mathsf{Proph}_1(p, v) \triangleq \exists vs.\ \mathsf{Proph}(p, vs) * (vs = [\,] \lor \mathit{head}(vs).2 = v)$$

With this, the specifications for one-shot prophecies can be derived from those for sequence prophecies. The only interesting part is in the proof of SEQUENCE-PROPHECY-SIMPLE-RESOLUTION, where $vs = [\,]$ will lead to a contradiction.

2.4 Atomic Prophecy Resolution

Another use-case of sequence prophecies is to predict the *interleaved sequence* of future actions by several threads. In particular, when many threads are racing to be the first to update some shared location, it can be important to know in advance *which thread will win* (*e.g.*, we will need this in §6).

Prophecy variables as we described them so far are insufficient for this use-case. We can use sequence prophecies to determine which thread will be the first to *resolve* a prophecy p, but the problem is that resolving p takes place in its own step of execution. To accurately predict the interleaving of atomic actions by multiple threads, we need a way to resolve p in the same step that we perform some other atomic action that we care about (*e.g.*, updating a shared location).

This is the purpose of the *atomic resolution operation*: $\texttt{Resolve}(e, p, w)$. This operation evaluates atomically to a value v if e evaluates to v atomically. Furthermore, during this atomic step, the prophecy variable p also gets resolved to the value (v, w), *i.e.*, the pair comprising the result of the underlying atomic operation together with the resolution value w.

We can give the following specification for the atomic resolution operation:

SEQUENCE-PROPHECY-RESOLUTION

$$\frac{\text{phys\_atomic(e)}}{\{\mathsf{Proph}(p, vs)\}\ \texttt{Resolve}(e, p, w)\ \{v.\ \exists vs'.\ vs = (v, w) :: vs' * \mathsf{Proph}(p, vs')\}}$$

In fact, the simple prophecy resolution operation we have seen so far is derived from the following atomic resolution operation (using the do-nothing atomic operation **skip**):

$$\texttt{Resolve}\ p\ \texttt{to}\ w \triangleq \texttt{Resolve}(\texttt{skip}, p, w)$$

Here, **skip** reduces to (), which explains why () appeared in the spec of **Resolve** p **to** w from §2.3.

3 SOUNDNESS OF PROPHECY VARIABLES

In this section, we justify the soundness of prophecy variables by adapting Iris's model of Hoare triples to handle them. In Iris, Hoare triples are not a primitive construct; rather, they are encoded in terms of a *weakest precondition* proposition as follows:

$$\{P\}\ e\ \{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\})$$

Here, the logical connective $-\!\!*$ is the *separating implication* (a.k.a. *magic wand*) from separation logic, and the $\Box$ connective is Iris's *persistence* modality, which is used to ensure that a Hoare triple (once proven) is a freely duplicable fact.

The weakest precondition proposition $\mathsf{wp}\ e\ \{\Phi\}$ says that if e reduces (in any number of steps) to e', then either e' is a value and the postcondition $\Phi(e')$ holds, or e' reduces further. As a consequence, $\mathsf{wp}\ e\ \{\Phi\}$ implies that e is safe (*i.e.*, never gets stuck in any execution). This intuitive understanding of weakest pre is formalized in a theorem called Adequacy, which we present in §3.4.

The central insight behind our model of prophecy variables is the following:

- To extend Iris's model of $\mathsf{wp}\ e\ \{\Phi\}$ to one that supports prophecies, we *parameterize* it by the sequence of future prophecy resolutions of e, and we use that parameter as the ground truth for modeling prophecy assertions.
- In the proof of the Adequacy theorem, we are given as input some reduction sequence Σ starting at e and ending at e'. Since prophecy resolutions are performed by actual operations during the execution of e, we can read off the sequence of values that each prophecy variable will be resolved to, just by looking at the reduction sequence Σ. We can then use that information to instantiate the new parameter in the model of $\mathsf{wp}\ e\ \{\Phi\}$.

In the rest of this section, we make this central insight precise. We first present the operational semantics of our language (§3.1). Afterwards, we briefly introduce a specific variant of the authoritative resource algebra (§3.2), which is used in modeling both the heap and prophecy resources. We then present the definition of weakest preconditions (§3.3) and formally state the Adequacy statement (§3.4). We conclude with the presentation of the Erasure theorem for ghost code (§3.5).

3.1 Operational Semantics

The essence of the operational semantics of our language is captured by the small-step head reduction relation ($\rightarrow_{\mathsf{h}}$), several rules of which are given at the top of Fig. 6. The reduction rules are of the form $(e_1, \sigma_1) \rightarrow_{\mathsf{h}} (e_2, \sigma_2, \vec{e}_f, \vec{\kappa})$, which should be read in the following way: expression e_1 in state σ_1 steps to expression e_2 in state σ_2, spawning threads computing the expressions in $\vec{e}_f$, and making the observations $\vec{\kappa}$ (used for recording prophecy resolutions). The state σ consists of a pair of a heap (represented as a finite map associating locations to values) and a set of already used names for prophecy variables. We write $\sigma.1$ and $\sigma.2$ for the first projection (the heap) and the

Examples of rules for the head reduction step relation

$$(\mathbf{fork}\ \{e\}, \sigma) \to_{\mathsf{h}} ((), \sigma, [e], \epsilon)$$
$$(\mathbf{CmpX}(\ell, v_1, v_2), \sigma \uplus \{\ell \leftarrow w\}) \to_{\mathsf{h}} ((w, \mathbf{true}), \sigma \uplus \{\ell \leftarrow v_2\}, \epsilon, \epsilon) \quad (v_1 = w)^\dagger$$
$$(\mathbf{CmpX}(\ell, v_1, v_2), \sigma \uplus \{\ell \leftarrow w\}) \to_{\mathsf{h}} ((w, \mathbf{false}), \sigma \uplus \{\ell \leftarrow w\}, \epsilon, \epsilon) \quad (v_1 \neq w)^\dagger$$

† To remain realistic, comparison requires one of the compared values to be "word-sized" (*e.g.*, integers, but not pairs).

$$(\mathbf{NewProph}, \sigma) \to_{\mathsf{h}} (p, \sigma \uplus \{p\}, \epsilon, \epsilon) \qquad \frac{(e, \sigma) \to_{\mathsf{h}} (v, \sigma, \vec{e}, \vec{\kappa})}{(\mathbf{Resolve}(e, p, w), \sigma) \to_{\mathsf{h}} (v, \sigma, \vec{e}, \vec{\kappa} +\!\!+ [(p, (v, w))])}$$

Evaluation contexts, per-thread and thread-pool reduction relation

$$K ::= K(e) \mid v(K) \mid \mathbf{ref}(K) \mid !K \mid K \leftarrow e \mid v \leftarrow K \mid \mathbf{Resolve}(e_1, e_2, K) \mid \mathbf{Resolve}(e, K, v) \mid \ldots$$

$$\frac{(e_1, \sigma_1) \to_{\mathsf{h}} (e_2, \sigma_2, \vec{e}_f, \vec{\kappa})}{(K[e_1], \sigma_1) \to (K[e_2], \sigma_2, \vec{e}_f, \vec{\kappa})} \qquad \frac{(e_1, \sigma_1) \to (e_2, \sigma_2, \vec{e}_f, \vec{\kappa})}{(T_1 +\!\!+ e_1 :: T_2, \sigma_1) \to_{\mathsf{tp}} (T_1 +\!\!+ e_2 :: T_2 +\!\!+ \vec{e}_f, \sigma_2, \vec{\kappa})}$$

$$(T_1, \sigma_1) \to_{\mathsf{tp}}^* (T_1, \sigma_1, [\,]) \qquad \frac{(T_1, \sigma_1) \to_{\mathsf{tp}} (T_2, \sigma_2, \vec{\kappa}_1) \qquad (T_2, \sigma_2) \to_{\mathsf{tp}}^* (T_3, \sigma_3, \vec{\kappa}_2)}{(T_1, \sigma_1) \to_{\mathsf{tp}}^* (T_3, \sigma_3, \vec{\kappa}_1 +\!\!+ \vec{\kappa}_2)}$$

Fig. 6. Elements of the operational semantics.

second projection (the used prophecy variable names), respectively. For conciseness, we use the notation $\sigma \uplus \{\ell \leftarrow v\}$ for $(\sigma.1 \uplus \{\ell \leftarrow v\}, \sigma.2)$, and notation $\sigma \uplus \{p\}$ for $(\sigma.1, \sigma.2 \uplus \{p\})$.

The main novelty here, compared to previous languages considered in Iris, is the presence of the list of observations $\vec{\kappa}$. It records events of interest during evaluation, which in our case are prophecy resolutions: the rule for $\mathbf{Resolve}(e, p, w)$ adds $(p, (v, w))$ to the list of observations, where v is the result of evaluating expression e atomically (*i.e.*, in exactly one step).

The "compare and exchange" instruction $\mathbf{CmpX}(\ell, v_1, v_2)$ atomically compares v_1 with the current value w of location ℓ (requiring one of v_1 and w to be word-sized, to make atomic comparison realistic), and if they are equal then stores v_2 in ℓ. The instruction returns a pair of the previous value w and a boolean indicating whether the exchange took place.

To complete the definition of the operational semantics, the per-thread reduction relation ($\to$), the thread-pool reduction relation ($\to_{\mathsf{tp}}$), and the transitive closure of the thread-pool reduction relation are given by the rules at the bottom of Fig. 6. The per-thread reduction step corresponds to a head step reduction performed under an evaluation context. The thread-pool reduction non-deterministically picks a thread in the thread-pool and runs it for a single step, adding all the threads spawned to the thread pool. The transitive closure of the thread-pool reduction ($\to_{\mathsf{tp}}^*$) accumulates all the observations of individual thread-pool steps.

3.2 Authoritative Resource Algebra

One of the distinguishing features of Iris is its very general notion of (user-defined) ownership, based on a form of resource algebras called *cameras* [Jung et al. 2018]. Most notably, the heap and the associated $\ell \mapsto v$ proposition (asserting the ownership of a location ℓ containing value v) are encoded using the *authoritative resource algebra* that we will introduce shortly. Another instance of the same resource algebra is used for prophecy variables, which intuitively have their own (distinct) heap. The proposition $\mathsf{Proph}(p, vs)$ thus plays a role similar to $\ell \mapsto v$: it asserts the exclusive ownership of a prophecy variable p with "value" (really: future resolutions) vs.

The authoritative resource algebra contains two kinds of elements:[2] *authoritative* elements (denoted $\bullet M$), which carry a finite map, and *fragment* elements (denoted $\circ \{i \leftarrow e\}$), which carry a single key-value pair. The intuition is that there is only one authoritative element, and that every existing fragment should agree (*i.e.*, be compatible) with it. This is reflected in the following rule:

$$\boxed{\bullet M}^{\gamma} * \boxed{\circ \{i \leftarrow e\}}^{\gamma} \Rightarrow \{i \leftarrow e\} \in M \qquad \text{(AUTH-AGREE)}$$

The notation $\boxed{r}^{\gamma}$ denotes ownership of an element r of the resource algebra where the "ghost name" γ is used to distinguish different instances of resource algebras. Later we will use arbitrary but globally fixed names γ_{HEAP} and γ_{PROPH} for the two resource algebras involved in the semantics of weakest preconditions.

Owned resources can be *updated* using *view shifts*, $P \Rrightarrow Q$. The authoritative resource algebra can be updated by allocating new fragments and by updating the value of a fragment; in each case the authoritative part needs to be updated accordingly:

$$i \notin \mathrm{dom}(M) * \boxed{\bullet M}^{\gamma} \Rrightarrow \boxed{\bullet M \uplus \{i \leftarrow e\}}^{\gamma} * \boxed{\circ \{i \leftarrow e\}}^{\gamma} \qquad \text{(AUTH-ALLOC)}$$

$$\boxed{\bullet M \uplus \{i \leftarrow e_1\}}^{\gamma} * \boxed{\circ \{i \leftarrow e_1\}}^{\gamma} \Rrightarrow \boxed{\bullet M \uplus \{i \leftarrow e_2\}}^{\gamma} * \boxed{\circ \{i \leftarrow e_2\}}^{\gamma} \qquad \text{(AUTH-UPDATE)}$$

The ownership of locations and prophecy variables are both defined in terms of a fragment of an authoritative algebra as follows:

$$\ell \mapsto v \triangleq \boxed{\circ \{\ell \leftarrow v\}}^{\gamma_{\text{HEAP}}} \qquad \mathsf{Proph}(p, vs) \triangleq \boxed{\circ \{p \leftarrow vs\}}^{\gamma_{\text{PROPH}}}$$

As we will see in the next section, the corresponding authoritative parts of these resource algebras are used in the definition of weakest preconditions.

3.3 Model of Weakest Preconditions

The definition of weakest preconditions is given below, with our extensions for supporting prophecy variables marked in blue.[3]

$$\begin{aligned}
&\mathrm{wp}\ e_1\ \{\Phi\} \triangleq \text{if } e_1 \in \mathit{Val} \text{ then } \Phi(e_1) \text{ else} && \textit{(return value)}\\
&\quad \forall \sigma_1, \vec{\kappa}_1, \vec{\kappa}_2.\, S(\sigma_1, \vec{\kappa}_1 +\!\!+ \vec{\kappa}_2) \Rrightarrow *\\
&\qquad \mathrm{reducible}(e_1, \sigma_1) * && \textit{(progress)}\\
&\qquad \forall e_2, \sigma_2, \vec{e}_f.\, ((e_1, \sigma_1) \rightarrow (e_2, \sigma_2, \vec{e}_f, \vec{\kappa}_1)) \Rrightarrow * \\
&\qquad\quad S(\sigma_2, \vec{\kappa}_2) * \mathrm{wp}\ e_2\ \{\Phi\} * *_{e \in \vec{e}_f} \mathrm{wp}\ e\ \{\mathsf{True}\} && \textit{(preservation)}
\end{aligned}$$

$$S(\sigma, \vec{\kappa}) \triangleq \boxed{\bullet \sigma.1}^{\gamma_{\text{HEAP}}} * \exists \Pi.\, \boxed{\bullet \Pi}^{\gamma_{\text{PROPH}}} * \mathrm{dom}(\Pi) = \sigma.2 * \forall \{p \leftarrow vs\} \in \Pi.\, vs = \mathrm{filter}(p, \vec{\kappa}) \qquad \textit{(state interpretation)}$$

Let us ignore the new parts for the moment. If e_1 is a value, then the weakest precondition simply requires the postcondition to hold. If, on the other hand, e_1 is not a value, then e_1 should be *reducible* under any state σ_1 (consisting only of the heap for now) that matches the existing $\ell \mapsto v$ assertions. This connection is made by the state interpretation S: the authoritative element is tied to σ_1, and thus we know that the $\ell \mapsto v$, which are fragments of the same resource algebra, must agree (AUTH-AGREE). Furthermore, for any $(e_2, \sigma_2, \vec{e}_f)$ that (e_1, σ_1) steps to, we should be able to update resources so that the new state still satisfies S. Finally, the expressions e_2 that we step to should still satisfy the weakest precondition w.r.t. Φ while the spawned threads should satisfy a weakest precondition with the trivial postcondition (we do not care about the end result of spawned threads).

[2] We are only considering a particular instance of authoritative resource algebra here.

[3] For lack of space, we will gloss over some details (*e.g.*, the use of guarded recursion and mask-changing view shifts); these details are orthogonal to the present work and can be found in Jung et al. [2018] and the Coq development of Iris.

The key in supporting prophecies is to change the state interpretation S to be a predicate not only on the heap but also on the sequence of *future* prophecy resolutions. Now this predicate additionally asserts authoritative ownership of a mapping Π, which maps each prophecy variable to the sequence of its future resolutions (here, the function $\mathrm{filter}(p, \vec{\kappa})$ removes all resolutions not corresponding to p from $\vec{\kappa}$). Since the second argument $\vec{\kappa}$ of the state interpretation predicate is the sequence of *future* prophecy resolutions, each step s of computation removes from $\vec{\kappa}$ the prophecies resolved by s—hence, in the definition above, the state interpretation is $S(\sigma_1, \vec{\kappa}_1 +\!\!+ \vec{\kappa}_2)$ before the step and $S(\sigma_2, \vec{\kappa}_2)$ after the step, with $\vec{\kappa}_1$ being the prophecies resolved in this step.

Proving SEQUENCE-PROPHECY-CREATION. We will now use the definition of weakest preconditions above to derive the prophecy creation rule (involving the **NewProph** ghost instruction):

SEQUENCE-PROPHECY-CREATION
$$\{\mathsf{True}\}\ \mathtt{NewProph}\ \{p.\ \exists vs.\ \mathrm{Proph}(p, vs)\}$$

The expression **NewProph** is not a value. Let us assume that we have $S(\sigma, \vec{\kappa})$ for some state σ and future resolutions $\vec{\kappa}$. Obviously (see Fig. 6), **NewProph** is reducible under σ. Thus, let us assume that $(\mathtt{NewProph}, \sigma) \to (p, \sigma \uplus \{p\}, \epsilon, \epsilon)$ for some fresh p not appearing in $\sigma.2$. Hence, we have to show:

$$S(\sigma, \vec{\kappa}) \Rrightarrow S(\sigma \uplus \{p\}, \vec{\kappa}) * \mathrm{wp}\ p\ \{p.\ \exists vs.\ \mathrm{Proph}(p, vs)\}$$

Or equivalently, by unfolding the definition of weakest precondition and some simplification:

$$S(\sigma, \vec{\kappa}) \Rrightarrow S(\sigma \uplus \{p\}, \vec{\kappa}) * \exists vs.\ \mathrm{Proph}(p, vs) \tag{2}$$

The domain of the authoritative prophecy map Π existentially quantified in S should always match the set of declared prophecy variables. Here, we are declaring a new prophecy variable and hence we should update Π accordingly. Since we know that p is fresh, *i.e.*, $p \notin \sigma.2$, we know that it also does not appear in the domain of Π. Therefore, we can update Π using AUTH-ALLOC; we simply need to pick a sequence of values for the future resolutions of p. This however is completely determined by the second line of the definition of S: it must be $\mathrm{filter}(p, \vec{\kappa})$, just as we would intuitively expect. Hence, we obtain (2) with $vs \triangleq \mathrm{filter}(p, \vec{\kappa})$. □

Proving SEQUENCE-PROPHECY-SIMPLE-RESOLUTION. We now sketch the proof of the resolution rule:

SEQUENCE-PROPHECY-SIMPLE-RESOLUTION
$$\{\mathrm{Proph}(p, vs)\}\ \mathtt{Resolve}\ p\ \mathtt{to}\ w\ \{\exists vs'.\ vs = ((), w) :: vs' * \mathrm{Proph}(p, vs')\}$$

We start out with ownership of $S(\sigma, \vec{\kappa}_1 +\!\!+ \vec{\kappa}_2)$, and we know that $(\mathtt{Resolve}\ p\ \mathtt{to}\ w, \sigma) \to ((), \sigma, \epsilon, \vec{\kappa}_1)$, which means that $\vec{\kappa}_1 = [(p, ((), w))]$. From $\mathrm{Proph}(p, vs)$ and AUTH-AGREE, we have $\{p \leftarrow vs\} \in \Pi$, where Π is the authoritative prophecy map, and hence $vs = \mathrm{filter}(p, (p, ((), w)) :: \vec{\kappa}_2) = ((), w) :: vs'$, where $vs' = \mathrm{filter}(p, \vec{\kappa}_2)$. Finally, using AUTH-UPDATE, we can simultaneously update the state interpretation to $S(\sigma, \vec{\kappa}_2)$ and the prophecy assertion to $\mathrm{Proph}(p, vs')$, as required. □

3.4 Adequacy

The adequacy theorem states that if the weakest precondition of a program e is provable with respect to a *pure* postcondition ϕ, then e is safe with respect to that postcondition, which we write as $\mathit{Safe}_\phi(e)$. A predicate is pure if it can be written at the meta-level (*e.g.*, Coq), and does not use any Iris logic connectives. The proposition $\mathit{Safe}_\phi(e)$ asserts that when e is executed, all the involved threads are safe. Moreover, if the main thread (*i.e.*, e itself) reduces to a value, then the postcondition ϕ holds for that value.

$$\mathit{Safe}_\phi(e) \triangleq \forall T, \sigma, \vec{\kappa}.\ ([e], \varnothing) \to^*_{\mathsf{tp}} (e' :: T, \sigma, \vec{\kappa}) \Rightarrow \mathit{proper}_\phi(e', \sigma) \wedge \forall e_t \in T.\ \mathit{proper}_{\mathsf{True}}(e_t, \sigma)$$

$$\mathit{proper}_\psi(e, \sigma) \triangleq (e \in \mathit{Val} \wedge \psi(e)) \vee \mathrm{reducible}(e, \sigma)$$

Theorem 3.1 (Adequacy). *Let e be an expression and ϕ be a pure predicate over values.*

$$\mathit{If} \ \vdash \mathit{wp}\, e\, \{\phi\} \ \ \mathit{then} \ \ \mathit{Safe}_\phi(e).$$

To prove Adequacy, we are given up front a full reduction sequence $([e], \varnothing) \to^*_{\mathrm{tp}} (e' :: T, \sigma, \vec{\kappa})$ starting at the expression e under the empty heap $\varnothing$. Crucially, this execution trace gives us access to the full sequence $\vec{\kappa}$ of prophecy resolutions that will be made in this execution. We can thus construct an initial state interpretation $S((\varnothing, \varnothing), \vec{\kappa})$ with an empty heap and an empty prophecy map Π (no prophecy variable has been created yet). We can then instantiate wp $e\,\{\phi\}$ with our state interpretation and step through our execution trace until we eventually obtain wp $e'\,\{\phi\}$ and wp $e_t\,\{\mathsf{True}\}$ for all $e_t \in T$, and our state interpretation becomes $S(\sigma, \epsilon)$. To conclude the proof, we then simply use the "progress" and "return value" parts of the weakest preconditions.

3.5 Erasure

As we have seen, in order to use our prophecies, one must not only work with prophecy assertions in the logic, but also instrument the code with prophecy-related "ghost" operations. Intuitively, it is expected that such ghost operations do not affect the operational behavior of programs and should be safely erasable. To make this formal, we first define a function *erase*, which eliminates ghost operations from programs. The interesting cases of the definition of *erase* are the following:

$$\begin{aligned}
\mathit{erase}(\texttt{NewProph}) &\triangleq (\lambda_.\ \text{☣})() \\
\mathit{erase}(\texttt{Resolve}(e_1, e_2, e_3)) &\triangleq \pi_1\,(\pi_1\,((\mathit{erase}(e_1), \mathit{erase}(e_2)), \mathit{erase}(e_3))) \\
\mathit{erase}(\texttt{CmpX}(e_1, e_2, e_3)) &\triangleq \texttt{CmpX}(\mathit{erase}(e_1), \mathit{erase}(e_2), \mathit{erase}(e_3))
\end{aligned}$$

The erased `NewProph` still takes a step of computation to simplify the proof, but instead of a prophecy name it returns a special *poison* value ☣. This ensures that the erased program has the right behavior when a prophecy name is being compared with other values: comparing two prophecy names with each other is forbidden (they are not considered word-sized), while (real and erased) prophecy names are considered unequal to any word-sized value. For non-ghost operations such as `CmpX` and all omitted cases, the *erase* function just proceeds structurally.

We are now ready to state our Erasure theorem:

Theorem 3.2 (Erasure). *Let e be a program and ϕ be a pure predicate over values such that $\mathit{Safe}_\phi(e)$ holds. Then, $\mathit{Safe}_{\widehat{\phi}}(\mathit{erase}(e))$ where $\widehat{\phi}$ is as follows: $\widehat{\phi}(v) \triangleq \exists v'.\ \phi(v') \land v = \mathit{erase}(v')$.*

It is worth emphasizing that both Adequacy and Erasure pertain only to *whole (closed) programs* that have been proven safe unconditionally (*i.e.*, under trivial precondition True) and whose postconditions ϕ are pure. In particular, Erasure says nothing about preserving Hoare triples in general because erasure does not preserve them! Hoare triples may involve prophecy assertions, whose meaning is fundamentally tied to the prophecy resolutions in the code, so erasing prophecy resolutions from the program will not preserve the meaning of those assertions. But this is fine, because we only care about erasing prophecy code for whole programs that can actually execute.

4 LOGICAL ATOMICITY

We will now move on from the illustrative but contrived coin examples to showcase a much more interesting and sophisticated application of prophecy variables. In particular, we will show how prophecy variables enable us to prove Iris-style *logical atomicity*—a relative of *linearizability*—for certain tricky concurrent data structures.

To make the presentation concrete, we will show how to prove logical atomicity for the RDCSS data structure of Harris et al. [2002]. RDCSS is a key building block in the implementation of

another data structure, MCAS (multi-word CAS), which allows one to compare and exchange any number of machine words atomically. As explained in §1, RDCSS has already served as a key case study in prior work that attempted to integrate prophecy variables into Hoare logic [Vafeiadis 2008; Zhang et al. 2012], but previous accounts of its verification were unsatisfactory.

In this section, we describe the abstract semantics that RDCSS is supposed to provide, and we use this as high-level motivation for explaining the general concept of *logical atomicity* and how it is formalized in Iris. Then, in §5, we present the actual RDCSS implementation, along with an intuitive argument for its correctness which motivates the need for prophecies. Finally in §6, we explain in significant detail how we prove logical atomicity of RDCSS formally in Iris using prophecies.

4.1 RDCSS Semantics

To explain the operational behavior of RDCSS, let us for a moment pretend that we would just add it as a primitive to our programming language. So we would have a new language operation $\mathbf{RDCSS\_prim}(\ell_m, \ell_n, m_1, n_1, n_2)$ with the following reduction rule:

$$\begin{aligned}&(\mathbf{RDCSS\_prim}(\ell_m, \ell_n, m_1, n_1, n_2), \sigma \uplus \{\ell_m \leftarrow m, \ell_n \leftarrow n\})\\ &\quad\to_{\mathsf{h}} \quad (n, \sigma \uplus \{\ell_m \leftarrow m, \ell_n \leftarrow \big((m, n) = (m_1, n_1)\ ?\ n_2\ :\ n\big)\}, \epsilon, \epsilon)\end{aligned}$$

In other words, if m and n are the original values of ℓ_m and ℓ_n, then $\mathbf{RDCSS\_prim}(\ell_m, \ell_n, m_1, n_1, n_2)$ *atomically* compares them to m_1 and n_1, respectively, and if both of them compare equal, ℓ_n is updated to point to n_2. Either way, the old value n is returned by the operation.

4.2 Logical Atomicity: Who Needs It?

Of course, our programming language does not *actually* contain $\mathbf{RDCSS\_prim}$ as a primitive operation, and neither does real hardware. To actually use RDCSS in an implementation of MCAS, one has to somehow implement it on top of the existing primitives. In §5, we will present the code for an efficient implementation of RDCSS, which we will refer to as RDCSS (see Fig. 7). But before we can think about verifying RDCSS, we first need to have a clear idea of what specification we want to prove for it.

Naively, one might think that the following Hoare triple is a good specification for RDCSS:

$$\begin{aligned}&\{\ell_m \mapsto m * \ell_n \mapsto n\}\\ &\quad \mathsf{RDCSS}(\ell_m, \ell_n, m_1, n_1, n_2)\\ &\{n'.\, n' = n * \ell_m \mapsto m * \ell_n \mapsto \big((m, n) = (m_1, n_1)\ ?\ n_2\ :\ n\big)\}\end{aligned} \qquad \text{(RDCSS-SPEC-SEQ)}$$

This specification directly reflects the operational semantics: m and n are the initial values stored in the two locations, and if $m = m_1$ and $n = n_1$ then the value in ℓ_n is updated to n_2.

Unfortunately, while RDCSS-SPEC-SEQ would indeed be a valid specification for RDCSS (as well as for $\mathbf{RDCSS\_prim}$), it is not a terribly useful one. To see why, observe that we could prove the same specification for the following, *non-thread-safe* implementation of RDCSS:

$$\begin{aligned}\mathsf{RDCSS\_Seq}(\ell_m, \ell_n, m_1, n_1, n_2) \triangleq\ &\mathbf{let}\ m = !\ell_m;\\ &\mathbf{let}\ n = !\ell_n;\\ &\ell_n \leftarrow (\mathbf{if}\ m = m_1\ \&\&\ n = n_1\ \mathbf{then}\ n_2\ \mathbf{else}\ n);\\ &n\end{aligned}$$

Why is that? Because the precondition of RDCSS-SPEC-SEQ gives RDCSS *exclusive* ownership of the locations ℓ_m and ℓ_n at the beginning, and requires that the operation return exclusive ownership of these locations at the end. Given such a strong precondition, the non-thread-safe RDCSS_Seq behaves indistinguishably from the thread-safe RDCSS and the primitive $\mathbf{RDCSS\_prim}$. However, such

a spec is fundamentally *sequential*: it requires the client of RDCSS to give up exclusive ownership of the data structure in order to invoke the operation, thus prohibiting concurrent accesses to the data structure. To support concurrent accesses, we need to find a stronger, thread-safe spec, one which RDCSS will satisfy but RDCSS_Seq will not.

Towards a thread-safe, Hoare-style spec for RDCSS. The canonical, strong, thread-safe specification for concurrent data structures like RDCSS is *linearizability* [Herlihy and Wing 1990]: it says that even if multiple RDCSS operations are running interleaved with one another, they still behave the same as some linear sequence of **RDCSS_prim** operations. The problem with linearizability is that it operates *outside of Hoare logic*—it is expressed formally in terms of traces, whereas we want a property that we can gainfully employ when verifying clients of RDCSS *inside of Hoare logic*.

Enter *logical atomicity*. The point of logical atomicity (which we will define below) is to reflect the idea of linearizability into the logic of Iris, so that client verifications can make use of it—*i.e.*, so that clients can program against RDCSS but pretend for the purposes of verification that they are programming against **RDCSS_prim**.

This begs the question: what is so special about **RDCSS_prim** that clients would want to pretend that they are programming against it? The answer is *atomicity*: the key property that **RDCSS_prim** has, which RDCSS does not, is that it is *physically atomic*—*i.e.*, it completes in a single step of computation. And the reason, in turn, that we care about atomicity is that it enables clients to invoke **RDCSS_prim** when accessing an *invariant*.

Invariants are the central mechanism in concurrent separation logic for enforcing protocols on the use of shared resources by concurrently running threads. Resources governed by an invariant may be accessed using the *invariant rule*:[4]

$$\textsc{inv}\ \frac{\{R * P\}\ e\ \{v.\, R * Q(v)\} \qquad \text{phys\_atomic(e)}}{\boxed{R} \vdash \{P\}\ e\ \{v.\, Q(v)\}}$$

Here, $\boxed{R}$ asserts the existence of an invariant guarding the resources described by the assertion R. Once established, $\boxed{R}$ is a duplicable fact that can be freely shared between multiple threads so that they can apply INV in parallel. The rule itself is best read clockwise, beginning at the bottom-left (the precondition of the conclusion): starting with resources P, when one is reasoning about a physically atomic operation e, one can temporarily *gain* ownership of the shared resource described by R so long as one *returns* ownership of some (potentially modified) shared resource that still satisfies R when e is finished executing. One is left with $Q(v)$ in the postcondition of the conclusion, describing other resources that one gets to keep. The requirement that e be physically atomic ensures that, even if the invariant R is broken at some point *within* the verification of e, no other thread will notice so long as R holds before and after.

Because **RDCSS_prim** is physically atomic, we can use INV to verify programs that call **RDCSS_prim** on shared locations. For example, suppose we established an invariant R which asserted ownership of ℓ_m and ℓ_n together with the stipulation that ℓ_n pointed to an even number. In that case, the rule INV would enable multiple threads to invoke **RDCSS_prim** on these locations concurrently, so long as the **RDCSS_prim** operations only ever updated ℓ_n to an even number. In contrast, RDCSS is *not* physically atomic, so even in the presence of $\boxed{R}$, it would not be possible to use INV to verify concurrent calls to RDCSS on ℓ_m and ℓ_n.

To summarize: *the key advantage of physically atomic operations is that one can access invariants around them.* In concurrent separation logics, this is the only "special power" that physically atomic

[4]Technical details related to the "later" modality and to "namespaces" have been omitted for simplicity. They are not directly relevant to what we are discussing here, but the interested reader can refer to Jung et al. [2018].

operations have, but it is a big one. The goal of logical atomicity is to also grant this special power to operations like RDCSS that *behave* as if they were physically atomic, even though they are not.

Logically atomic triples. Formally, logical atomicity is encoded using *logically atomic Hoare triples*. We write them just like normal Hoare triples, but with angle brackets instead of curly braces. The key benefit of logically atomic triples (from the point of view of their clients) is expressed by the *logically atomic invariant rule*:

$$\textsc{logatom-inv}\quad \frac{\langle R * P\rangle\ e\ \langle v.\, R * Q(v)\rangle}{\boxed{R} \vdash \langle P\rangle\ e\ \langle v.\, Q(v)\rangle}$$

Compared to INV, this rule is almost the same, except there is no side condition of physical atomicity. The ideal specification for RDCSS thus looks as follows:

$$\begin{array}{l}\langle m, n.\, \ell_m \mapsto m * \ell_n \mapsto n\rangle\\ \quad \mathsf{RDCSS}(\ell_m, \ell_n, m_1, n_1, n_2)\\ \langle n'.\, n' = n * \ell_m \mapsto m * \ell_n \mapsto \big((m, n) = (m_1, n_1)\ ?\ n_2\ :\ n\big)\rangle\end{array} \qquad (\textsc{rdcss-spec-ideal})$$

This specification concisely expresses that RDCSS has the same pre- and postcondition as in our original sequential specification RDCSS-SPEC-SEQ, but moreover clients can verify concurrent invocations of RDCSS because they can use LOGATOM-INV to access invariants around it.

Notice the binders for m and n in the precondition: usually, free variables in a Hoare triple are interpreted as universally quantified, so that the client is free to instantiate them however they want when using the triple. But m and n are somewhat special in that during the execution of RDCSS (which is logically atomic but still takes many steps to execute), their values can actually *change*. To support this, logically atomic triples come with a special binder in the precondition. The technical details of this point do not matter so much for our presentation in this paper, so we refer the reader to Jung et al. [2015] for details.

Separating abstract from physical state. Sadly, RDCSS-SPEC-IDEAL turns out to be too restrictive for the implementation. As is common when verifying complex data structures, we have to separate the *physical state* that makes up the data structure implementation from the *abstract state* that is currently represented by the data structure. For the RDCSS implementation of Fig. 7, the current abstract value of location ℓ_n is not stored literally; instead a more complex physical state is used to coordinate all threads involved in the concurrently running RDCSS operations. In the specification, this means we cannot work with $\ell_n \mapsto n$ as that exposes the underlying physical value stored at ℓ_n. Instead we use an *abstract predicate* $\mathsf{RState}(\ell_n, n)$ to express that the current *abstract* value stored at ℓ_n is n. How exactly that is represented physically is considered an implementation detail. In particular, this "seals off" ℓ_n in the sense that clients of RDCSS can only interact with ℓ_n through the operations provided by the RDCSS library: using normal loads or stores would require ownership of $\ell_n \mapsto n$ which is never exposed to clients.

For ℓ_m, it turns out that abstract and physical state coincide, so we do not need an abstract predicate there. This is useful for clients to know as it means that they can use any other heap operations (loads, stores, **CmpX**) on ℓ_m, even in parallel with RDCSS—a property that MCAS, built on top of RDCSS, relies on. The specification thus becomes:

$$\begin{array}{l}\langle m, n.\, \ell_m \mapsto m * \mathsf{RState}(\ell_n, n)\rangle\\ \quad \mathsf{RDCSS}(\ell_m, \ell_n, m_1, n_1, n_2)\\ \langle n'.\, n' = n * \ell_m \mapsto m * \mathsf{RState}(\ell_n, (m, n) = (m_1, n_1)\ ?\ n_2\ :\ n)\rangle\end{array} \qquad (\textsc{rdcss-spec})$$

4.3 Proving Logically Atomic Triples

Although the idea of logically atomic triples is conceptually simple, their formalization remains one of the most technically advanced constructions in the Iris logic. Fortunately, we need not go into the full gory details of this construction in order to get across the main ideas about how these triples are proved and how prophecy variables can play an instrumental role in proving them.

To understand how logically atomic triples are proved, let us recall how they are used. Once we have proven the logically atomic triple RDCSS-SPEC, clients of RDCSS will be able (thanks to rule LOGATOM-INV) to access invariants around applications of the RDCSS operation in their code. In order for it to be sound for clients to do this, there must be a *single physical step* during the execution of RDCSS at which the abstract states of ℓ_n and ℓ_m are transformed as specified by RDCSS-SPEC. This point is known as the *linearization point*: it is the point in time when, abstractly, RDCSS "happens".

Consequently, when we prove $\langle P \rangle\ e\ \langle Q \rangle$, we should intuitively think of P and Q as really being pre- and postconditions not to e in its entirety but *to e's linearization point*. In other words, unlike the proof of a standard Hoare triple, the proof of $\langle P \rangle\ e\ \langle Q \rangle$ is not given ownership of precondition P at the beginning of e's execution, with the mandate to transform it into ownership of Q by the end. Rather, the transformation from P to Q is only allowed to take place during a single atomic step, namely the linearization point.

The question now is how this requirement of atomically transitioning from P to Q is reflected in the logic. To this end, the Iris encoding of logically atomic triples relies on an *atomic update*: a token $\mathsf{AU}_{P,Q}(\Phi)$, which represents both the *right* and the *obligation* to transition in one physical step from an abstract state satisfying P to an abstract state satisfying Q. In the proof, we will use this atomic update at the linearization point in order to "commit" the abstract effect of the operation. At that point, the atomic update is *consumed* and the assertion Φ is given back as a *receipt*.

The role of the receipt Φ is to enforce the "obligation" part of this contract, as shown by the following introduction rule for logically atomic triples:[5]

$$\textsc{logatom-intro}\quad \frac{\forall \Phi.\ \{\mathsf{AU}_{P,Q}(\Phi)\}\ e\ \{\Phi\}}{\langle P \rangle\ e\ \langle Q \rangle}$$

To prove the logically atomic triple in the conclusion, we have to prove an ordinary Hoare triple with a different pre and post. The precondition is an atomic update permitting us to transition the abstract state from P to Q (at the linearization point). The postcondition is simply the receipt Φ, which means we *must* consume the atomic update from the precondition at some point: since Φ is universally quantified, picking a linearization point is the only way to obtain a proof of it.

4.4 Summary of Logical Atomicity

Before going further, let us review the main points about Iris-style logical atomicity:

- *Logically atomic triples* are useful because they allow us to grant the same "special power" to concurrent method calls that we otherwise only grant to physically atomic operations, namely that clients who call those methods can access invariants around them.
- Proving logical atomicity of an operation involves identifying the *linearization point*, a single step during the operation when it *abstractly* takes place.
- To prove a logically atomic triple $\langle P \rangle\ e\ \langle Q \rangle$ in Iris, we start out with ownership of an atomic update $\mathsf{AU}_{P,Q}(\Phi)$, and we must finish with ownership of a receipt Φ. The former provides the

[5] For simplicity, we have omitted details about the bound variables appearing in logically atomic triples, as they are orthogonal to our focus on prophecy variables. For details, see Jung et al. [2015].

```
NewRDCSS(n) ≜ ref(inl(n));

rec Get(ℓ_n) ≜
  match !ℓ_n with
    inl(n)        ⇒ n
  | inr(ℓ_descr) ⇒ Complete(ℓ_descr, ℓ_n); Get(ℓ_n)
  end;

Complete(ℓ_descr, ℓ_n) ≜
let (ℓ_m, m_1, n_1, n_2, p) = !ℓ_descr;
let tid = NewGhostId;
let m = !ℓ_m;
let n_new = if m = m_1 then n_2 else n_1;
Resolve(CmpX(ℓ_n, inr(ℓ_descr), inl(n_new)), p, tid);
()
```

```
1  RDCSS(ℓ_m, ℓ_n, m_1, n_1, n_2) ≜
2  let p = NewProph;
3  let ℓ_descr = ref(ℓ_m, m_1, n_1, n_2, p);
4  rec rdcss_inner() =
5    let (v, b) = CmpX(ℓ_n, inl(n_1), inr(ℓ_descr));
6    match v with
7      inl(n)      ⇒
8        if b then
9          Complete(ℓ_descr, ℓ_n); n_1
10       else   n
11   | inr(ℓ'_descr) ⇒
12       Complete(ℓ'_descr, ℓ_n); rdcss_inner()
13   end;
14 rdcss_inner()
```

Fig. 7. The RDCSS implementation.

right to execute the abstract transition from P to Q, and the latter enforces that the atomic update was used (exactly once) during the proof, namely at the linearization point.

5 UNDERSTANDING THE RDCSS IMPLEMENTATION

Now that we have formalized the specification for RDCSS using logical atomicity, we take a closer look at the actual implementation in Fig. 7 (ignoring the ghost code for now). Our goal here is to understand the correctness argument of the implementation at an intuitive level, focusing on the identification of the linearization point, as this will motivate why we need prophecies.

To use the RDCSS implementation, a client would first call NewRDCSS(n) to create an "RDCSS location" ℓ_n with initial value n. The current value of such a location can be read using Get(ℓ_n). Our focus is on the key operation RDCSS($\ell_m, \ell_n, m_1, n_1, n_2$) for an RDCSS location ℓ_n and a location ℓ_m, which (modulo minor technical details, as we will see) implements the specification RDCSS-SPEC.

The key to verifying that RDCSS satisfies RDCSS-SPEC is to develop an invariant which connects the physical state of ℓ_n with its abstract state. In the following, we will first explain how the physical state of ℓ_n maintained by this implementation is used to coordinate concurrent threads, and we will give some first insights into how the invariant relates the physical and abstract states of ℓ_n (we will make this invariant explicit in §6), before diving into the code.

The physical state of ℓ_n. The RDCSS implementation guarantees that at any point in time, at most one RDCSS operation is *active*. Only once the active operation has been successfully completed can another pending RDCSS operation be executed. If a thread wants to perform an RDCSS operation but another operation is already active, then it tries to *help* complete the active operation before attempting to perform its own operation again.

Concretely, an RDCSS location ℓ_n has two states, *inactive* and *active*:

- The *inactive* state is represented by the physical value **inl**(n). This indicates that the abstract value at ℓ_n is n, and there is currently no RDCSS operation ongoing.
- The *active* state is represented by the physical value **inr**(ℓ_{descr}). The *descriptor* ℓ_{descr} stores all the information needed about the active operation that is currently ongoing: it points to a

tuple of the form (ℓ_m, m_1, n_1, n_2), indicating that the operation currently being performed is RDCSS$(\ell_m, \ell_n, m_1, n_1, n_2)$ and that the abstract value at ℓ_n when this operation first became active was n_1 (*i.e.*, the physical value at ℓ_n was **inl**(n_1)). Crucially, each operation is associated with a *unique immutable* descriptor, so we can compare two descriptors to see if they refer to the same operation. This uniqueness prevents the well-known ABA problem, but it also means descriptors cannot be reused (our implementation implicitly relies on them being garbage-collected).

The RDCSS *implementation.* We now turn our attention to the code of RDCSS$(\ell_m, \ell_n, m_1, n_1, n_2)$[6].

Before anything interesting happens, the thread t_0 executing RDCSS first has to register the new operation it is attempting to perform as *active*. To this end, t_0 allocates a new descriptor ℓ_{descr} with all the required information in line 3, and runs $rdcss_{inner}$ to repeatedly attempt to register this as the active descriptor using **CmpX** in line 5.

If the value v previously stored at ℓ_n is of the form **inr**(ℓ'_{descr}), then the **CmpX** must have failed, and there is already an active operation on ℓ_n initiated by another thread. In that case, t_0 tries to help finish the active operation by calling Complete in line 12, and then loops to try registering its own descriptor again.

Otherwise, v is of the form **inl**(n). In this case the **CmpX** might either have succeeded or failed. If it failed (*i.e.*, b is **false**), then that means $n \neq n_1$. As a result, the current RDCSS operation behaves like a no-op, and this failed **CmpX** is its linearization point.

In the remaining case, **CmpX** actually succeeded (*i.e.*, b is **true**). In this case, t_0 has established that the ℓ_n part of the comparison succeeded (the abstract value at ℓ_n is n_1 since the physical value is **inl**(n_1)), but m_1 still needs to be compared with the value stored at ℓ_m. So, the successful **CmpX** has registered t_0's descriptor as the active one for this RDCSS location and t_0 calls Complete to try to finish the operation.

While this descriptor is active, every subsequent thread which tries to register its descriptor at the same location ℓ_n will fail and will enter the race to help finish t_0's operation by calling Complete on t_0's descriptor. One of the threads competing in this race will win it by successfully updating ℓ_n in line 20 before t_0's Complete call returns.

Let us now consider the code for Complete. First, we read the descriptor in line 16 to obtain the information for the operation specified by the caller. We then read the value m at location ℓ_m in line 18 and compare m with m_1 in line 19. If the values match, then we attempt to update ℓ_n in line 20 to **inl**(n_2), signalling that the corresponding RDCSS operation updates ℓ_n from abstract value n_1 to n_2. Otherwise, we attempt to update ℓ_n back to **inl**(n_1) (the value it had when the operation was first registered); the RDCSS operation is a no-op in this case.

Identifying the linearization point. To verify RDCSS-SPEC in the latter case where t_0's operation is active, we must find the linearization point, which should occur during t_0's call to Complete. This is the point at which the atomic update associated with t_0's operation is consumed, and the abstract states of ℓ_n and ℓ_m are updated *atomically* according to the semantics of **RDCSS_prim**.

But where should this linearization point be? The first thing to note is that the linearization point is not necessarily executed by t_0 itself—it will be executed by whichever thread t_w wins the race to help complete t_0's operation (which may or may not be t_0). And that is fine so long as it occurs *during* t_0's execution of RDCSS. (This phenomenon is known in the concurrency literature as "helping".)

[6]There are some minor differences compared to the implementation of RDCSS in Harris et al. [2002]. In the original version, the descriptor is taken as an argument instead of being allocated by RDCSS like we did. Adjusting our implementation and proof to handle that is straightforward. Moreover, they distinguish descriptors (active state) from the inactive state by checking the least significant bits, while we use injections to avoid having to formalize bit-level representations of values.

$S1$ $t_w : \mathbf{let}\,(\ell_m, m_1, n_1, n_2) = !\ell_{descr}$; // value at ℓ_m is m_1, value at ℓ_n is $\mathbf{inr}(\ell_{descr})$
$S2$ $t_w : \mathbf{let}\, m = !\ell_m$; // $m = m_1$
$S3$ $t_w : \mathbf{let}\, n_{new} = \mathbf{if}\, m = m_1\, \mathbf{then}\, n_2\, \mathbf{else}\, n_1$; // $n_{new} = n_2$
$S4$ $t_2 : \ell_m \leftarrow m_1'$ // value at ℓ_m is now m_1'
$S5$ $t_w : \mathbf{CmpX}(\ell_n, \mathbf{inr}(\ell_{descr}), \mathbf{inl}(n_{new}))$ // t_w's **CmpX** succeeds

Fig. 8. Snippet of an execution trace involving two threads t_w and t_2, where t_w is executing `Complete`(ℓ_{descr}, ℓ_n) (winning the race) and t_2 interferes by updating ℓ_m.

The obvious (but wrong) choice, then, is for the linearization point to be when the winning thread t_w uses **CmpX** to successfully update the physical value at ℓ_n in line 20. Intuitively, the reason this does not work is that the read of ℓ_m in line 18 and the subsequent **CmpX** in line 20 are not performed atomically. Since ℓ_m is subject to arbitrary interference—it can be updated by concurrent operations at any time—there is no reason to believe that ℓ_m has the same value at line 20 that it did at line 18.

To show this more concretely, consider the possible execution trace in Fig. 8. The trace shows the execution steps of `Complete` by t_w, where thread t_2 interferes by writing to ℓ_m in between when t_w executes line 18 and line 20 of `Complete`. In execution step $S5$ of the trace, t_w successfully updates ℓ_n to $\mathbf{inl}(n_2)$ because this is consistent with the value m_1 which t_w read at ℓ_m in execution step $S2$. But due to t_2's interference, the abstract state of ℓ_m when t_w performs the update is m_1', which is different from the value t_w read. This means t_w cannot consume the atomic update in step $S5$, because the value of ℓ_m dictates that the abstract state of ℓ_n cannot be updated here from n_1 to n_2 according to the specification of `RDCSS` (since $m_1 \neq m_1'$). Hence the only option is for the linearization point to be in step $S2$ when t_w reads ℓ_m, since it is the only point where we can be certain that the value read by t_w is consistent with ℓ_m.

As we will formally show in §6, this observation can be generalized. In every scenario, it is possible to set things up such that the winner t_w consumes the atomic update at the instant when t_w reads ℓ_m in line 18. Intuitively, it is "safe" to pick this point as the linearization point, because we know that the physical value of ℓ_n will not be changed (*i.e.*, it stays in-sync with t_w's view) before t_w successfully executes the update in line 20.

The problem, of course, is that the linearization point occurs at line 18 *only for the winning thread* t_w. The winning thread should consume t_0's atomic update at that point; the other threads in the race should not. But in the Iris proof, how are we supposed to know up front which thread is the winner? We only learn that later on when verifying line 20. This is known in the concurrency literature as a *future-dependent linearization point*, and in order to handle it, we will need a *prophecy*.

6 FORMALLY VERIFYING RDCSS USING PROPHECIES

In the previous sections, we (1) gave a high-level specification for the correctness of `RDCSS` using logically atomic triples, and (2) explained the working principle of the implementation shown in Fig. 7 with a focus on determining its *linearization point*, the step during its execution at which its abstract action takes place. In this section, we will now talk about how to actually verify the desired specification in Iris, and why that requires a powerful form of prophecy variables.

6.1 Using Prophecies for Proving RDCSS

As we saw in §4.3, when we are proving a logically atomic triple and we reach the program step where the linearization point occurs, we must *know* that this is actually the linearization point because we need to consume the atomic update associated with the operation at that point.

In the case of proving RDCSS($\ell_m, \ell_n, m_1, n_1, n_2$), however, this is problematic. Recall that after the thread t_0 succeeds in registering its descriptor with the **CmpX** operation in line 5, the linearization point is when the thread t_w, which *wins* the race to complete the operation, reads ℓ_m in line 18. The problem, of course, is that at the point when t_w reads ℓ_m, it does not yet know that *it* is going to be the winning thread (*i.e.*, the thread whose **CmpX** update will succeed in line 20), and therefore in the proof of Complete we do not know at this point whether the atomic update must be consumed or not. Naturally, the solution to this problem is to use a prophecy variable: we will set up a prophecy so that each thread knows whether its future update will succeed.

But how do we set up this prophecy? A first idea is to create a new prophecy at the beginning of Complete. This is precisely what Vafeiadis [2008] did, as we mentioned in §1.1. However, there is a flaw in this approach: for each thread t_i trying to complete the active operation we would have a separate prophecy p_i predicting the outcome of the corresponding **CmpX** update. If p_i predicts that t_i's update will succeed, then the linearization point is when t_i reads ℓ_m. Hence in the proof of t_i's Complete operation, we will consume the atomic update at line 18 if the prophecy said that our future **CmpX** will succeed. To do so, however, we must ensure that no *other* thread has already consumed the atomic update for the same active operation, and it is not clear how to establish this. It requires making sure that, for all the threads competing to complete the operation, only one of them will have a prophecy that predicts its own success. But there was no mechanism to enforce this condition in Vafeiadis's proof, and thus (as we confirmed with him) it is incorrect.

The problem is that, when we learn about the contents of the prophecy, we might get a "spurious" result that says something obviously wrong (*e.g.*, predicting multiple winners). However, in such a case, we cannot immediately derive a contradiction—that would be a cyclic argument, as the correctness of the remainder of the proof relies on this information. Instead, we have to "play along" with this clearly incorrect spurious prophecy until such a time when reality contradicts what the prophecy predicted. Only *then* can we obtain a contradiction. In fact, this concern already came up *en passant* in §2.2, where typed prophecies had to use a "fake value" in case the prophesied value had the wrong type—a spurious prophecy. There too, the contradiction was only reached later, when the prophecy got resolved with a value of the right type, refuting what had been prophesied.

To get around this spurious prophecy issue we take a fundamentally different approach. Instead of creating one prophecy variable per thread trying to complete the active operation, we create a *single* prophecy variable per active operation. The prophecy is created by the thread that registers the RDCSS operation in the first place (see the **NewProph** statement in line 2 of RDCSS), and it is stored in the operation's descriptor. The prophecy predicts the sequence in which the threads trying to complete the operation will execute the update in Complete—and clearly, the first one in that sequence is the winner. Having a single prophecy controlling all the threads fixes the problem in Vafeiadis [2008]'s argument. Note, however, that our use of prophecy variables is inherently "unstructural": one thread prophesies the updates of other threads. Hence, this proof cannot be conducted using the "structural" prophecy variables of Zhang et al. [2012] (as described in §1.1).

For our approach to work, we need a way for threads to identify themselves. This is achieved in Complete by using the **NewGhostId** statement[7] to create a fresh "thread identifier". In the logic this creates a ghost resource GID(*tid*) which is exclusive, ensuring that *tid* is globally unique. Then, in line 20, we can resolve the prophecy to the thread identifier *tid* that we have generated.

[7] **NewGhostId** can be encoded using **NewProph**. Hence the erasure theorem ensures that it has no effect on the original code.

$$\{\mathsf{True}\}\ \texttt{NewRDCSS}(n)\ \{\ell_n.\ \mathsf{RState}(\ell_n, n) * \mathsf{IsRDCSS}(\ell_n)\}$$

$$\mathsf{IsRDCSS}(\ell_n) \vdash \langle n.\ \mathsf{RState}(\ell_n, n)\rangle\ \texttt{Get}(\ell_n)\ \langle v.\ v = n * \mathsf{RState}(\ell_n, n)\rangle$$

$$\mathsf{IsRDCSS}(\ell_n) \vdash \left\langle m, n.\quad \ell_m \mapsto m * \mathsf{RState}(\ell_n, n) \right\rangle \texttt{RDCSS}(\ell_m, \ell_n, m_1, n_1, n_2) \left\langle v.\quad v = n * \ell_m \mapsto m * \mathsf{RState}(\ell_n, (m, n) = (m_1, n_1)\,?\,n_2 : n) \right\rangle$$

Fig. 9. Specification for the RDCSS operations.

6.2 RDCSS Specification

In the next section, we are going to highlight some important details of the proof that the RDCSS implementation of Fig. 7 indeed satisfies the expected specification. To this end, we first complete RDCSS-SPEC to a specification of all three RDCSS operations as shown in Fig. 9.[8]

The specification of the operations involves two assertions, $\mathsf{IsRDCSS}(\ell_n)$ and $\mathsf{RState}(\ell_n, n)$, the definitions of which are not exposed to the user. We have already seen $\mathsf{RState}(\ell_n, n)$ when we first discussed RDCSS-SPEC: it asserts ownership of RDCSS location ℓ_n and also records that it is in abstract state n. This is comparable to the "maps-to" assertion ($\ell \mapsto v$), but specific to RDCSS locations. The new assertion, $\mathsf{IsRDCSS}(\ell_n)$, states that ℓ_n is an RDCSS location and is thus governed by the RDCSS invariant (which will be presented later). $\mathsf{IsRDCSS}(\ell_n)$ is freely duplicable ("persistent" in Iris lingo): it can be used arbitrarily often and is not subject to the usual "may be used only once" rule of separation logic. This lets us call arbitrary RDCSS operations as often as we want.

As a side point: note that unlike the proof of Vafeiadis [2008], our setup allows dynamically growing the set of RDCSS locations over time (by calling NewRDCSS).

6.3 Proving the RDCSS Specification

We are now going to give some details of the proof of the RDCSS specification that was informally described in §6.1. As introduced in §4.3, to prove the logically atomic triple for RDCSS given in Fig. 9, we must first apply LOGATOM-INTRO. We must hence prove[9]

$$\mathsf{IsRDCSS}(\ell_n) \vdash \left\{\mathsf{AU}_{\mathsf{RD}_{pre},\mathsf{RD}_{post}}(\Phi)\right\}\ \texttt{RDCSS}(\ell_m, \ell_n, m_1, n_1, n_2)\ \{\Phi\}$$

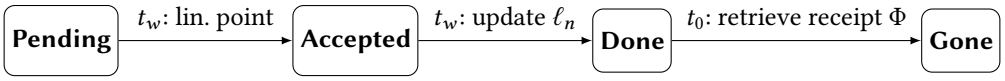
for an arbitrary Φ, where RD_{pre} and RD_{post} respectively denote the pre- and postconditions of the logically atomic triple. The only way to prove this Hoare triple is to consume the atomic update $\mathsf{AU}_{\mathsf{RD}_{pre},\mathsf{RD}_{post}}(\Phi)$ at the linearization point, by updating the abstract state of ℓ_n as specified by RD_{pre} and RD_{post}.

The RDCSS and descriptor protocols. For this proof, we need a way to capture the protocols which track the relation between the abstract state of ℓ_n and the physical state as partially described in §5. We already informally explained the *RDCSS protocol* distinguishing between ℓ_n being inactive and active. The inactive state is simple: ℓ_n points to $\texttt{inl}(n)$, where n is its abstract state.

If ℓ_n is active—*i.e.*, it points to $\texttt{inr}(\ell_{descr})$—the descriptor ℓ_{descr} is governed by a separate *descriptor protocol*, depicted in Fig. 10. At the point when the descriptor is first registered in line 5, the protocol starts out in the Pending state, and the abstract state of ℓ_n in that state is n_1. When the winning

[8] In the formal specification of RDCSS in Coq, we do not use the normal points-to assertion for ℓ_m in the precondition. Instead we use a "GC" points-to assertion which also guarantees that ℓ_m is never deallocated. This is required so that ℓ_m can always be read safely, but we suppress this detail here since it is completely orthogonal to the main proof.

[9] We ignore the return value here for the sake of simplifying the presentation.



Pending: ℓ_n points to **inr**(ℓ_{descr}), and linearization point (LP) has not yet occurred

Accepted: LP has occurred (consuming AU and producing Φ), but ℓ_n still points to **inr**(ℓ_{descr})

Done: ℓ_n has been **CmpX**'d to next inactive (**inl**) state

Gone: t_0's `Complete` has returned, and t_0 has retrieved receipt Φ from the protocol

Fig. 10. Descriptor protocol governing descriptor ℓ_{descr} which is registered by the thread t_0. Here, t_w is the winning thread that will complete t_0's operation with a successful **CmpX**.

$$\mathsf{IsRDCSS}(\ell_n) \triangleq \boxed{\mathsf{Inactive}(\ell_n) \vee \mathsf{Active}(\ell_n)}$$

$$\mathsf{Inactive}(\ell_n) \triangleq \exists n.\, \ell_n \mapsto \mathbf{inl}(n) * \mathsf{RState}^{\bullet}(\ell_n, n)$$

$$\mathsf{Active}(\ell_n) \triangleq \exists \ell_{descr}, q, m_1, n_1, n_2, p, t_w, \Phi, \mathsf{ITok}, \cdots .\, \mathsf{IsDescr}(\ell_{descr}, \ell_n, n_1, p, t_w, \Phi, \mathsf{ITok}) * \ell_n \overset{0.5}{\longmapsto} \mathbf{inr}(\ell_{descr}) * \ell_{descr} \overset{q}{\mapsto} (\ell_m, m_1, n_1, n_2, p) * \cdots$$

Fig. 11. $\mathsf{IsRDCSS}(\ell_n)$ specifies the knowledge of the RDCSS invariant encoding the RDCSS protocol which governs ℓ_n. Only a partial definition is shown (as hinted at by $\cdots$).

thread t_w (which might or might not be the same as the thread t_0 that registered this descriptor) reads ℓ_m in line 18, the linearization point occurs—*i.e.*, the atomic update (AU) is consumed, the abstract state of ℓ_n is updated, and the receipt Φ is produced—and the protocol moves to Accepted. From that point onwards, ℓ_{descr} remains registered until t_w successfully updates ℓ_n back to an inactive state and the protocol reaches the Done state. Finally, once the original thread t_0 returns from `Complete`, the protocol moves to the Gone state, and t_0 retrieves its receipt Φ.

Since these protocols govern shared resources, we encode them using Iris's invariants, which were presented in §4.2. Essentially, this encoding models a protocol as an invariant that maintains a disjunction of assertions, where each disjunct corresponds to one possible state in the protocol and stores the shared resources we need to track in that state. In the proof, when we need to inspect the state of a protocol, we access the corresponding invariant using INV to temporarily obtain the resources guarded by it. Depending on what private resources we owned before accessing the invariant, we may be able to rule out certain states (if the shared resources governed by the invariant in those states are incompatible with our private resources). Finally, at the end of the same execution step we either move the protocol to a new state or leave the state unchanged, in either case giving back the shared resources as required by the corresponding disjunct of the invariant.

The definition of the *RDCSS invariant*, which encodes the RDCSS protocol for ℓ_n, is given in Fig. 11. In both states, the invariant contains ownership of ℓ_n, ensuring that threads can access ℓ_n concurrently (we explain this ownership in more detail below). When a thread t_0 registers its descriptor ℓ_{descr} successfully, it simultaneously moves the state of ℓ_n's RDCSS protocol from Inactive to Active. The resources $\mathsf{Active}(\ell_n)$ required to complete this state change serve to set up the communication between all the threads trying to complete t_0's operation.

In particular, one of the resources t_0 must provide as part of $\mathsf{Active}(\ell_n)$ is the knowledge of the *descriptor invariant* (Fig. 12), which encodes the descriptor protocol for ℓ_{descr} that we depicted in Fig. 10. In order to initialize the descriptor invariant in the Pending state, t_0 must transfer multiple resources to the descriptor invariant, including: the prophecy resource $\mathsf{Proph}(p, \mathit{tids})$

$$\mathsf{IsDescr}(\ell_{descr}, \ell_n, n_1, p, t_w, \Phi, \mathsf{ITok}) \triangleq \boxed{\begin{array}{l}\mathsf{Pend}(\ell_{descr}, \ell_n, n_1, p, t_w, \Phi) \lor \mathsf{Acc}(\ell_{descr}, \ell_n, p, t_w, \Phi) \lor \\ \mathsf{Done}(\ell_{descr}, p, t_w, \Phi) \lor \mathsf{Gone}(\ell_{descr}, p, t_w, \mathsf{ITok})\end{array}}$$

$$\mathsf{Pend}(\ell_{descr}, \ell_n, n_1, p, t_w, \Phi) \triangleq \begin{array}{l}\mathsf{AU}_{\mathsf{RD}_{pre},\mathsf{RD}_{post}}(\Phi) * (\exists tids.\, \mathsf{Proph}(p, tids) * \mathsf{IsWinner}(t_w, tids)) * \\ \ell_n \overset{0.5}{\longmapsto} \mathbf{inr}(\ell_{descr}) * \mathsf{RState}^{\bullet}(\ell_n, n_1) * \cdots\end{array}$$

$$\mathsf{Acc}(\ell_{descr}, \ell_n, p, t_w, \Phi) \triangleq \begin{array}{l}\Phi * (\exists tids.\, \mathsf{Proph}(p, tids) * \mathsf{IsWinner}(t_w, tids)) * \\ \ell_n \overset{0.5}{\longmapsto} \mathbf{inr}(\ell_{descr}) * \mathsf{GID}(t_w) * \cdots\end{array}$$

$$\mathsf{Done}(\ell_{descr}, p, t_w, \Phi) \triangleq \Phi * (\exists tids.\, \mathsf{Proph}(p, tids)) * \mathsf{GID}(t_w) * \cdots$$

$$\mathsf{Gone}(\ell_{descr}, p, t_w, \mathsf{ITok}) \triangleq \mathsf{ITok} * (\exists tids.\, \mathsf{Proph}(p, tids)) * \mathsf{GID}(t_w) * \cdots$$

$$\mathsf{IsWinner}(t_w, tids) \triangleq (tids \neq [\,] \land head(tids).2 \in GIDSet) \Rightarrow t_w = head(tids).2$$

Fig. 12. $\mathsf{IsDescr}(\ell_{descr}, \ell_n, \cdots)$ specifies the knowledge of the descriptor invariant encoding the descriptor protocol governing descriptor ℓ_{descr}. In this case ℓ_{descr} stores a tuple $(\ell_m, m_1, n_1, n_2, p)$ and hence identifies an RDCSS operation $\mathsf{RDCSS}(\ell_m, \ell_n, m_1, n_1, n_2, p)$. Only a partial definition is shown (as hinted at by $\cdots$).

which identifies the winner t_w of t_0's operation, the atomic update $\mathsf{AU}_{\mathsf{RD}_{pre},\mathsf{RD}_{post}}(\Phi)$ needed at the linearization point, and the resources for the physical and abstract ownership of ℓ_n. We will now explain the role that each of these resources play in the protocol, before showing some key steps of the proof.

Setting up the prophecy resource. Each thread t_i trying to complete an operation needs to know, before reading ℓ_m, who is the winning thread t_w for which the update in line 20 is going to succeed. This will decide whether t_i should consume the atomic update when it reads ℓ_m. To support this reasoning, t_0 (the initiator of the operation) creates a prophecy p in line 2 which predicts the sequence in which the threads helping to complete the operation will execute the update in line 20. The first thread in this sequence is the winning thread t_w.

t_0 transfers its prophecy resource $\mathsf{Proph}(p, tids)$ to the descriptor invariant in Pending, which can be used by helping threads when they need to do prophecy resolution. Since at this point, as well as in the Accepted state, p has not yet been resolved (the winner has not yet executed the update), t_w must still be the first thread in the current sequence $tids$ predicted by the prophecy. This property is expressed by $\mathsf{IsWinner}(t_w, tids)$, whose definition additionally accounts for the possibility that $tids$ could contain spurious values.

Note that the prophecy resource $\mathsf{Proph}(p, \_)$ is kept around in all states of the descriptor invariant, so that any thread competing in the race will be able to perform its prophecy resolution.

Communicating the atomic update. The only way for t_0 to obtain the receipt Φ is if the winner t_w consumes the atomic update when it reads ℓ_m, since that is the linearization point for t_0's operation. This means t_0 must somehow transfer its atomic update to t_w which in turn transfers the receipt back to t_0. t_0 initiates this transfer over the descriptor invariant by giving up its atomic update to the invariant in the Pending state when registering its descriptor. At the linearization point, t_w gets hold of the atomic update from the invariant and consumes it, obtaining the receipt Φ in the process. t_w then moves the protocol to the Accepted state by putting the receipt into the invariant, where it remains until the transition from Done to Gone occurs.

How can t_0 now retrieve the receipt? For this final step in the transfer, we use two ingredients. First, we introduce a unique *initiator token* ITok, which t_0 creates prior to establishing the descriptor

invariant, and which it owns privately until its `Complete` operation is finished. So long as t_0 owns ITok privately, we can rule out that the protocol is in state Gone, since in that state the descriptor invariant owns ITok, and the token cannot be owned by both t_0 and the invariant at the same time. Second, we will prove a specification for `Complete` (see below) which guarantees in its postcondition that the descriptor protocol is either Done or Gone. So, when t_0's call to `Complete` returns, t_0 can use ITok to conclude that the protocol must be exactly in the Done state. At that point, it can transition the protocol to the Gone state by trading ITok for the receipt Φ.

Tracking the abstract and physical state of ℓ_n. As already mentioned, one of the key ingredients in proving `RDCSS` correct is to track the abstract and physical state of ℓ_n. We track the abstract state using the resource $\mathsf{RState}^\bullet(\ell_n, n)$[10], expressing that the abstract state of ℓ_n is n.

In the Inactive state, the RDCSS invariant specifies that the physical state (expressed using the points-to assertion) is $\mathbf{inl}(n)$ and that the abstract state is n. In the Active state, the RDCSS invariant only specifies that the physical state is $\mathbf{inr}(\ell_{descr})$ for some descriptor ℓ_{descr}.

In the latter case, the physical state is expressed using the *fractional points-to assertion* [Boyland 2003] $\ell_n \overset{0.5}{\longmapsto} \mathbf{inr}(\ell_{descr})$. Fractional ownership permits one to distribute ownership of ℓ_n (the sum of all the fractions is 1 and a non-zero fraction is sufficient for reading). We keep the other half of the ownership in the Pending and Accepted states of the descriptor protocol. This expresses that in those states the descriptor described by this protocol is still the active descriptor pointed to by ℓ_n. Note that we use a similar technique for sharing access to ℓ_{descr}, keeping fractional ownership of ℓ_{descr} in the RDCSS invariant so that it can be safely read in `Complete`.

Furthermore, as we explained above, once the descriptor is registered, the abstract value of ℓ_n stays the same until the protocol is moved from Pending to Accepted (*i.e.*, until the linearization point happens). This is expressed by $\mathsf{RState}^\bullet(\ell_n, n_1)$ in the Pending state. Note that when the registered descriptor is in the Accepted state, it is the only point in time where we do not explicitly track in any invariant what the abstract value of ℓ_n is. The reason is that only the winner knows the abstract value at this point! We will see how this is reflected in the proof.

Controlling transitions using ghost resources. It is important that only certain threads can trigger specific transitions in the descriptor protocol. For example, the winner t_w should be able to prove at the linearization point that the protocol must be in the Pending state, since only t_w should be allowed to transition to Accepted. Otherwise the winner would not be able to get hold of the atomic update which it must consume at the linearization point. We achieve this using ghost resources.

As an example, let us consider the transition from Pending to Accepted. When we create the thread identifier *tid* in line 17, we obtain the ghost resource $\mathsf{GID}(tid)$. This ghost resource is exclusive, *i.e.*,

$$\mathsf{GID}(tid) * \mathsf{GID}(tid) \vdash \mathsf{False}$$

If we are the winner (*i.e.*, $t_w = tid$), then we obtain $\mathsf{GID}(t_w)$. As long as we keep private ownership of $\mathsf{GID}(t_w)$, we can prove that the descriptor protocol must be in the Pending state, using the same kind of reasoning (described above) that t_0 uses when establishing that the protocol is not in the Gone state. We use similar ghost resources to control the other transitions, but have suppressed the details of these resources in Fig. 12.

[10] $\mathsf{RState}^\bullet(\ell_n, n)$ is not the same resource as $\mathsf{RState}(\ell_n, n')$ used in the specification, although the two of them together imply that $n = n'$. We need this differentiation to control how the abstract state is updated: $\mathsf{RState}^\bullet$ is owned by the invariant, and RState is owned by the client. Only when applying the atomic update, when we momentarily own *both* $\mathsf{RState}^\bullet(\ell_n, n)$ and $\mathsf{RState}(\ell_n, n')$ at the same time, can we update *both* simultaneously to $\mathsf{RState}^\bullet(\ell_n, n'')$ and $\mathsf{RState}(\ell_n, n'')$ for some new value n''. This is a standard approach in Iris, which can be implemented using the authoritative RA (see §3.2).

Now that we have understood the main elements in the RDCSS and descriptor invariants, we highlight some important steps in the proof, for the interesting case where we succeed in registering our descriptor.

A specification for `Complete`. In line 9, right after t_0 registers its operation, t_0 invokes `Complete` to finish it. We would like a specification for `Complete` that can be used compositionally, not only by the initiator t_0 of the operation, but also by the threads helping the operation. With this in mind, we prove the following specification for `Complete`:

$$\mathsf{IsRDCSS}(\ell_n), \mathsf{IsDescr}(\ell_{descr}, \ell_n, n_1, p, t_w, \Phi, \mathsf{ITok}) \quad \vdash \quad \begin{array}{l} \{\ell_{descr} \overset{q}{\mapsto} (\ell_m, m_1, n_1, n_2, p)\} \\ \quad \texttt{Complete}(\ell_{descr}, \ell_n) \\ \{\mathsf{IsDoneOrGone}(\Phi, \mathsf{ITok})\} \end{array}$$

The preconditions a client must provide to use this specification are fairly straightforward: we only need read access to the descriptor, so any fraction q suffices. The interesting part is the postcondition $\mathsf{IsDoneOrGone}(\Phi, \mathsf{ITok})$, which asserts that the protocol governing ℓ_{descr} is either Done or Gone.[11]

$\mathsf{IsDoneOrGone}(\Phi, \mathsf{ITok})$ is a persistent, freely duplicable assertion, which reflects the fact that the protocol can never move to a previous state. As we mentioned above, the important point is that the initiator of the RDCSS operation, t_0, can use this assertion to exchange its ownership of ITok for the receipt Φ and finish its proof. The reason we do not put Φ into the postcondition directly is that such a specification would only be provable for t_0, not for the other threads helping to complete the operation—they do not own the necessary ITok.

In the proof of `Complete`, right after creating a thread identifier *tid* in line 17, we make a case distinction on whether *tid* is the winner t_w or not (recall that we know the winner by inspecting the prophecy resource in the descriptor invariant):

Case 1: tid is the winner t_w. In this case, while reading ℓ_m[12] we can establish that the descriptor protocol must be Pending (using the ghost resource $\mathsf{GID}(t_w)$ as explained above). Since the descriptor invariant provides $\mathsf{RState}^\bullet(\ell_n, n_1)$ in this state, we know that the abstract state of ℓ_n is n_1. Hence we can consume the atomic update (also provided by the invariant) by updating the abstract state depending on the value read in ℓ_m: if it is m_1, then update to n_2, otherwise keep it at n_1. We then move the protocol to Accepted by putting the obtained receipt Φ into the invariant and *keeping* the resource for the abstract state, namely $\mathsf{RState}^\bullet(\ell_n, n_{new})$ (where n_{new} is the correct new value for ℓ_n).

Finally, we attempt to update ℓ_n to $\texttt{inl}(n_{new})$, which we expect will succeed, since *tid* is the winner (but we have to prove now that the update will succeed!). At this point we access the RDCSS and the descriptor invariant. Using additional ghost resources (not shown) we can prove that the descriptor protocol must still be in Accepted and hence we know that the invariant guarantees that ℓ_{descr} is still registered at ℓ_n. This means the update will succeed and we can move the RDCSS invariant back to Inactive by giving up $\mathsf{RState}^\bullet(\ell_n, n_{new})$, which is in-sync with the physical value of ℓ_n at this point.

In the same step, we also move the descriptor protocol to Done and can thus establish the postcondition of `Complete`.

[11] For Iris aficionados, we encode this using a view shift, *i.e.*, $\mathsf{IsDoneOrGone}(\Phi, \mathsf{ITok}) \triangleq \Box(\mathsf{ITok} \Rrightarrow \Phi)$.

[12] Note that, since we are consuming the atomic update here, we can justify reading ℓ_m via RD_{pre}, which provides the required ownership of ℓ_m.

Case 2: tid is not the winner. In this case, we are not responsible for executing the linearization point, so we do not do anything special when reading ℓ_m.[13] We then attempt to update ℓ_n in line 20 (which we expect should fail, since we are not the winner).

At this point, we again inspect the RDCSS and descriptor invariants. Assume the descriptor protocol is either Done or Gone (as we expect, since the winner must have moved it at least to Done). In this case, we can prove that ℓ_{descr} is not registered anymore at ℓ_n and hence we can show that the update fails, which means we need not reason about a change in the physical state of ℓ_n and we can easily establish the postcondition of `Complete`.

Otherwise, contrary to our expectation, the descriptor protocol is not Done or Gone yet. This means that the winner has not yet updated ℓ_n and hence our **`CmpX`** should succeed. In this case the descriptor invariant asserts (via IsWinner(t_w, *tids*)) that if the next prophesied value is a thread identifier, then it must be the winner t_w. Therefore, after resolving the prophecy to *tid*, we learn that *tid* must be t_w. But since we assumed *tid* is not the winner, we have arrived at a contradiction! □

This completes the proof of logical atomicity for RDCSS. Despite the complexity of the argument, our formal mechanization of the proof [Jung et al. 2019] takes just 450 lines of Coq (ignoring comments), largely thanks to the powerful tactics of the Iris proof mode [Krebbers et al. 2017, 2018].

7 RELATED AND FUTURE WORK

Abadi and Lamport [1988, 1991] introduced the idea of prophecy variables as a complement to history variables. They observed that both kinds of auxiliary variables were in general necessary to prove that a more concrete *state machine* S_1 implements (*i.e.*, has a subset of the observable behaviors of) a more abstract one S_2 by means of a *refinement mapping* (a mapping of the states). Their main result is a *completeness* result: if S_1 implements S_2, then, under certain conditions, there exists a state machine S_1^{hp} obtained from S_1 by adding a history and a prophecy variable, such that there exists a refinement mapping from S_1^{hp} to S_2. They then stipulate a number of conditions which together ensure *soundness*, *i.e.*, that the instrumented S_1^{hp} has the same traces as the original state machine S_1. It is difficult to precisely compare their method to ours, since their framework is presented at the level of state machine refinement, and ours is specifically geared toward compositional reasoning in separation logic. That said, it seems that the way we extend our ghost state with a sequence of future observations could be seen as an instance of their notion of prophecy variable.

Vafeiadis [2008] was the first (to our knowledge) to propose the use of prophecy variables in Hoare logic, and for precisely the purpose that we have explored in this paper: verifying data structures with future-dependent linearization points. He does not give a formal justification, however, for their semantics in the context of Hoare logic, or how precisely to ensure that they are used soundly. Moreover, as we have explained in §6, the proof of his main example demonstrating the efficacy of prophecy variables—namely, RDCSS—is flawed. Nevertheless, as we have demonstrated in this paper, the idea to utilize prophecy variables for this purpose was fundamentally sound, and it has inspired others to follow suit. For example, García-Pérez et al. [2018] have recently used prophecy variables in formalizing linearizability arguments for Paxos-based systems.

As we explained in §1.1, Zhang et al. [2012] were the first (to our knowledge) to formalize any kind of prophecy variables in a Hoare logic. They restrict prophecies to be "structural", meaning that they must be resolved in the same syntactic scope in which they were created. Structural prophecies are enough to account for some data structures (*e.g.*, the "atomic snapshots" we have verified in our Coq development), but not others. In particular, all other data structures considered in this paper

[13] In this case, to justify reading ℓ_m, we rely on the fact that ℓ_m is modeled as a location that cannot be deallocated. See footnote 8 in §6.2.

(besides snapshots) use non-structural prophecy variables (where the prophecy may be resolved in a different thread than where it was created). Ironically, Zhang et al.'s main example to illustrate the power of structural prophecy variables is RDCSS, but their verification is buggy because it relies directly on the flawed proof of Vafeiadis [2008]. The flaw in his proof is precisely that his prophecies adhere to the structural discipline: each thread prophesies independently whether it will win the **CmpX** race, when in fact (as we argue in §5) the threads need to coordinate through a single non-structural prophecy variable to ensure there is only one winner.

Fu et al. [2010] avoid the need for history variables when reasoning about optimistic concurrency by using a Hoare logic where assertions describe the history trace leading up to the current state rather than just the current state. It seems plausible that the need for prophecy variables could be similarly eliminated by using assertions over the future trace, but this remains to be investigated in future work.

Turon et al. [2013] and Liang and Feng [2013] explore how to prove linearizability (or, in the case of Turon et al., the related notion of *contextual refinement*) for fine-grained concurrent data structures in separation logic. Both specifically tackle the verification of data structures with future-dependent linearization points (Liang and Feng in fact proved RDCSS, and Turon et al. proved a somewhat simplified version of RDCSS called conditional increment). To handle such data structures, both employ *speculation*, a mechanism that allows the auxiliary state to record multiple possible *logical* states the program could be in. Thus, for example, in the case of RDCSS, speculation would allow one, at the read of ℓ_m, to speculate whether the linearization point should happen there or not, producing two alternate "universes". Later on in the proof, once it became clear whether the **CmpX** succeeds, one could keep the "right" universe and discard the "wrong" one.

Speculation and prophecy variables are clearly related approaches in spirit, but different in detail. In a proof with speculation, one must reason about all possible future eventualities simultaneously, and can only discard some of them later on once it is clear they have become irrelevant. In our experience, this can sometimes complicate invariants in proofs. In a proof with prophecies, one is told the future up front, so there is no need to account for more than one possible future at a time. However, with prophecies, one must sometimes do explicit reasoning to show that unexpected "spurious" predictions imply a contradiction with what actually happens (*e.g.*, see the end of our RDCSS proof in §6), which brings its own complications.

Our initial motivation for exploring prophecy variables instead of speculation stemmed from our interest in proving Iris-style *logically atomic triples*, which were inspired by the atomic triples of the TaDA logic [da Rocha Pinto et al. 2014]. As we have seen in §4, logically atomic triples are stronger than linearizability or contextual refinement in that they internalize atomicity of concurrent operations in a Hoare-style specification, enabling clients to reason about such operations as if they were physically atomic. In proving logical atomicity of an operation, however, one must consume an "atomic update" resource (§4.3) when one reaches the linearization point; and given that atomic updates in Iris are represented using higher-order assertions involving magic wands and view shifts, it is unclear how one could speculate about a potential linearization point because Iris does not support speculative consumption of resources. In contrast, our account of prophecy variables constitutes a very lightweight extension to Iris, whose soundness is straightforward to establish.

Delbianco et al. [2017] achieve Hoare-style specifications for data structures with future-dependent linearization points, but *without* relying on prophecy variables. Their specifications are in terms of an auxiliary variable tracking the data structure's *history*, *i.e.*, the set of operations that have occurred on the data structure, partially ordered by their logical order. Specifically, they relate the data structure's history in the post-state of an operation to the one in the pre-state. This approach avoids having to identify a linearization point as it occurs, instead requiring only that a logical order be determined on overlapping operations by the time they terminate. However, while their

approach supports talking about the abstract state of the data structure at some past instant t at which the history has "settled", in contrast to our approach it does not support talking about the *current* abstract state. Their approach thus does not validate the logically atomic invariant rule (LOGATOM-INV), whereas our logically atomic specs do.

Even outside of Hoare logic, prophecy variables remain a fairly exotic technique, but there are a few verification systems we know of that have adopted them. Sezgin et al. [2010] use prophecy variables to integrate backward reasoning into QED, a static verification tool for verifying concurrent algorithms using *reduction* [Lipton 1975]. Cook and Koskinen [2011] propose an algorithm for automatically proving temporal properties (expressed in LTL) of programs given as a transition system. Their algorithm adds prophecy variables to the transition system so as to determinize it sufficiently to be able to apply the faster CTL methods. Lamport and Merz [2017] integrate prophecy variables into TLA+ and give a number of examples to illustrate their use in proving refinement mappings. They present one-prediction prophecy variables (akin to our one-shot prophecies), as well as array and data structure prophecy variables (which support multiple resolutions like our sequence prophecies but are different in detail). Their paper is intended more as a kind of tutorial guide for TLA+ users, and leaves formal soundness arguments to future work.

Since we introduced prophecy variables into Iris, they have already been put to good use. Penninckx et al. [2019] use them for giving abstract specifications of I/O behavior. In a paper appearing alongside ours in the present issue of PACMPL, de Vilhena et al. [2020] use them in an essential way in verifying a "local generic solver" in Iris. Interestingly, the latter application of prophecy variables has nothing at all to do with concurrency. As part of their development, de Vilhena et al. introduce rules for prophecy disposal and typed prophecies (not present in the original version of our paper). Our rules for typed prophecies in §2 loosely follow the style of theirs.

As far as we are aware, the idea of using separation logic and ownership to ensure the soundness of prophecy-based reasoning has not been proposed before and is a novel contribution of our paper.

Future work. In future work, we hope to address a limitation of our approach pertaining to logical atomicity. The entire *raison d'être* of logical atomicity is to be able to treat linearizable operations on a data structure as if they were physically atomic, *i.e.*, to grant them the same privileges that are afforded to physically atomic commands in concurrent separation logic. Unfortunately, our account of prophecy variables introduces a new operation that (at present) privileges physically atomic operations over logically atomic ones: namely, atomic prophecy resolution $\texttt{Resolve}(e, p, w)$ (described in §2.4). This mechanism allows one to resolve prophecy p to w in an atomic step together with any *physically* atomic operation e. That is sufficient for verifying logical atomicity of data structures implemented internally using physically atomic memory operations (like RDCSS implemented with **CmpX**). But in order to fulfill the promise of logical atomicity as described in the TaDA and Iris papers [da Rocha Pinto et al. 2014; Jung et al. 2015]—in particular, the ability to build logically atomic data structures from other logically atomic data structures—we would ultimately like to be able to attach prophecy resolution to any *logically atomic* operation as well. Our preliminary attempts to support such a mechanism suggest it will require a non-trivial extension of our present framework.

ACKNOWLEDGMENTS

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289), and in part by the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453 (VESSEDIA). Amin Timany is a postdoctoral fellow of the Flemish research fund (FWO).

REFERENCES

Martín Abadi and Leslie Lamport. 1988. The existence of refinement mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. 165–175. https://doi.org/10.1109/LICS.1988.5115

Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theor. Comput. Sci.* 82, 2 (May 1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

John Boyland. 2003. Checking interference with fractional permissions. In *SAS (LNCS)*, Vol. 2694. 55–72.

Byron Cook and Eric Koskinen. 2011. Making prophecies with decision predicates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 399–410. https://doi.org/10.1145/1926385.1926431

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In *ECOOP (LNCS)*, Vol. 8586. 207–231.

Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. 2020. Spy game: Verifying a local generic solver in Iris. *PACMPL* 4, POPL, Article 33 (Jan. 2020). http://gallium.inria.fr/~fpottier/publis/de-vilhena-pottier-jourdan-spy-game-2020.pdf

Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2017. Concurrent data structures linked in time. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 8:1–8:30. https://doi.org/10.4230/LIPIcs.ECOOP.2017.8

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: Compositional reasoning for concurrent programs. In *POPL*. 287–300.

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A mechanised relational logic for fine-grained concurrency. In *LICS*. 442–451.

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR (LNCS)*, Vol. 6269. 388–402.

Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. 2018. Paxos consensus, deconstructed and abstracted. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 912–939.

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A practical multi-word compare-and-swap operation. In *DISC*.

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *POPL*. 271–282.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28, e20 (Nov. 2018), 1–73. https://doi.org/10.1017/S0956796818000151

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2019. The future is ours: Prophecy variables in separation logic – Artifact. https://doi.org/10.5281/zenodo.3541918 (latest version available on paper website: https://plv.mpi-sws.org/prophecies/).

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*. 637–650.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–16:30.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217.

Leslie Lamport and Stephan Merz. 2017. Auxiliary variables in TLA+. *CoRR* abs/1703.05121 (2017). http://arxiv.org/abs/1703.05121

Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective auxiliary state for coarse-grained concurrency. In *POPL*. 561–574.

Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *PLDI*.

Richard J. Lipton. 1975. Reduction: A method of proving properties of parallel programs. *Commun. ACM* 18, 12 (Dec. 1975). https://doi.org/10.1145/361227.361234

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Informatica* 6, 4 (1976), 319–340. https://doi.org/10.1007/BF00268134

Willem Penninckx, Amin Timany, and Bart Jacobs. 2019. Specifying I/O using abstract nested hoare triples in separation logic. In *Proceedings of the 21st Workshop on Formal Techniques for Java-like Programs (FTfJP '19)*. ACM, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/3340672.3341118

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *LICS*. 55–74.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*. 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

Ali Sezgin, Serdar Tasiran, and Shaz Qadeer. 2010. Tressa: Claiming the future. In *VSTTE*.

Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *POPL*.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*. 691–707. https://doi.org/10.1145/2660193.2660243

Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification.* Ph.D. Dissertation. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-726.pdf

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A marriage of rely/guarantee and separation logic. In *CONCUR (LNCS)*, Vol. 4703. 256–271.

Zipeng Zhang, Xinyu Feng, Ming Fu, Zhong Shao, and Yong Li. 2012. A structural approach to prophecy variables. In *TAMC*. https://doi.org/10.1007/978-3-642-29952-0_12