Robust and Compositional Verification of Object Capability Patterns
(Long Version)

DAVID SWASEY, MPI-SWS∗
DEEPAK GARG, MPI-SWS∗
DEREK DREYER, MPI-SWS∗

In scenarios such as web programming, where code is linked together from multiple sources, object capability patterns (OCPs) provide an essential safeguard, enabling programmers to protect the private state of their objects from corruption by unknown and untrusted code. However, the benefits of OCPs in terms of program verification have never been properly formalized. In this paper, building on the recently developed Iris framework for concurrent separation logic, we develop OCPL, the first program logic for compositionally specifying and verifying OCPs in a language with closures, mutable state, and concurrency. The key idea of OCPL is to account for the interface between verified and untrusted code by adopting a well-known idea from the literature on security protocol verification, namely robust safety. Programs that export only properly wrapped values to their environment can be proven robustly safe, meaning that their untrusted environment cannot violate their internal invariants. We use OCPL to give the first general, compositional, and machine-checked specs for several commonly-used OCPs—including the dynamic sealing, membrane, and caretaker patterns—which we then use to verify robust safety for representative client code. All our results are fully mechanized in the Coq proof assistant.

1 INTRODUCTION

Suppose you have a mutable reference ℓ whose contents you care about, meaning that you want to impose some invariant on it (e.g., ℓ always points to an even number). Suppose further that you want to share access to ℓ with code you did not write and that you do not trust to preserve the invariant on ℓ. To ensure the invariant on ℓ is maintained, you therefore do not want to pass the untrusted code the reference ℓ directly. Instead, you might construct a read-only wrapper w as follows:

\[\text{readonly} \triangleq \lambda r. \lambda \_ . \! r\]

Here, readonly transforms a reference r into a thunk that, when applied, returns the current contents of r. The expression w applies readonly to our reference of interest ℓ, constructing a function for reading ℓ’s contents. You can now pass w to untrusted code without worrying about it corrupting your invariant on ℓ.

Wrappers like w are often called object capabilities, and the wrapper construction function readonly is a very simple example of an object capability pattern (OCP) (Miller et al. 2000). Although OCPs date back at least to the 1970s (Morris 1973), they have gained increased currency in recent years, both in new languages centered around object capabilities (Miller et al. 2000; Mettler et al. 2010; Spiessens and Roy 2004; Stiegler and Miller 2006) and in the context of web programming, where interfacing with untrusted code is commonplace. Web sandboxing systems like Yahoo!’s ADsafe (Crockford 2008; Politz et al. 2014) and Google’s Caja (Miller et al. 2008), for example, elaborate untrusted JavaScript source code into an ostensibly safe subset of JavaScript while introducing wrappers that attenuate access to JavaScript libraries. Caja relies on the so-called membrane pattern, which automatically wraps all objects crossing to untrusted, sandboxed code so that access to the underlying objects is appropriately restricted.

∗ Saarland Informatics Campus.
This work licensed under a Creative Commons Attribution 4.0 International License.
© 2017 David Swasey, Deepak Garg, and Derek Dreyer.
OCPs are believed to provide crucial security guarantees because, in a language like JavaScript, objects provide essentially no data abstraction on their own—the only way to enforce data abstraction is to hide private state in the environment of a closure (e.g., as $\ell$ was hidden in the environment of $\text{readonly} \ell$ in the example above). So OCPs, which use closures to mediate access to private state in a systematic way, are one of the few effective mechanisms available in a language like JavaScript for enforcing data abstraction in the presence of possibly malicious code.

Unfortunately, despite the ubiquity of OCPs in modern web programming, remarkably little attention has been paid to the question of what exactly are the security guarantees that such OCPs are supposed to provide, and how might we prove that they actually provide them. Even in the case of the extremely basic $\text{readonly}$ pattern shown above, it is not at all obvious what is the "right" formal specification for $\text{readonly}$. What, in particular, are the formal conditions on $\ell$ that are needed to guarantee that $\text{readonly} \ell$ can be “safely” shared with untrusted code? If $\ell$ merely points to an integer, no conditions may be necessary, but what if $\ell$ points to a closure or some other higher-order object? How do we know that giving readonly access to $\ell$ will not indirectly give untrusted code a way of gaining full access and violating important invariants maintained by the user of this OCP?

The most recent, state-of-the-art attempt to grapple with these types of questions is due to Devriese et al. (2016). They build a Kripke logical relations model for reasoning about object capabilities in a language with higher-order state, and they use their model to verify several concrete examples of capability-wrapped user code. In each case, they demonstrate that invariants on the private state of the user code are preserved even when the user code is shared with unknown attacker code. However, their model is limited in that it provides no way to compositionally specify what an OCP does, and it provides no clear specification of the general property that a piece of user code must satisfy in order to be safely shareable with untrusted code. Furthermore, they only consider very simple capability patterns, none as complex as, say, the aforementioned membrane pattern.

In this paper, we present OCPL (a Logic for OCPs), the first formal system for compositionally specifying and verifying the security guarantees provided by OCPs, in the context of a simple but representative programming language with higher-order functions, state, and concurrency. In contrast to prior work, OCPL enables one to reason modularly about both OCP implementations and user code that depends on them, and to specify a general property on user code that ensures such code can be safely shared with untrusted code without having its internal invariants violated. We use OCPL to reason about several commonly-used OCPs, including the dynamic sealing (sealer-unsealer), membrane, and caretaker patterns, and in so doing, provide the first formal explanation of what these OCPs achieve.

OCPL is a program logic derived from Iris, a recently developed framework for higher-order concurrent separation logic (Jung et al. 2015, 2016; Krebbers et al. 2017a). Iris was originally proposed as a very general logic with a few simple primitives, using which one can derive advanced proof principles from a variety of modern separation logics as needed. Moreover, Krebbers et al. (2017b) have recently developed a powerful new proof mode for Iris in the Coq proof assistant. The Iris proof mode enables one to carry out interactive, tactical, machine-checked proofs about programs in the Iris logic (embedded in Coq) in much the same way as one normally carries out interactive tactical proofs when working in the meta-logic of Coq itself. By virtue of building OCPL on top of Iris, we inherit the flexibility of the existing Iris framework, as well as its support for mechanizing proofs about programs in Coq. All our results and examples are, in fact, fully verified in Coq.\footnote{Despite the name, object capabilities are not fundamentally limited to object-oriented programming languages.\footnote{The Coq formalization is available online (OCPL 2017).}}
The key idea in OCPL is how it characterizes the interface between verified user code and untrusted code, via the concept of a “low-integrity value” (or low value for short) adapted from the literature on verification of security protocols (Abadi 1999). Roughly speaking, a low value is a value that can be safely shared with untrusted code, such as the closure wrappers returned by OCPs. More precisely, a low value is a value from which no code can possibly extract a direct reference to private state. To formalize this notion, we employ a logical relation, which is easy to define using Iris’s built-in support for guarded recursive predicates. Then, with the idea of low values in hand, it becomes clear how to specify when a piece of user code can be safely linked with untrusted code—namely, when the only values passed back and forth between them are low values.

To make things concrete, let us return to our motivating example. Using the notion of low values, we can prove the following specification for the readonly pattern:

$$\forall \ell. \{ \top \} !\ell \{ x. \text{lowval} x \} \Rightarrow \{ \top \} \text{readonly} \ell \{ f. \text{lowval} f \}$$

The premise of this spec says that, in order to apply readonly to a location \(\ell\), we must first prove that dereferencing \(\ell\) always produces a low value. (This makes sense because once we pass the read-only wrapper to untrusted code, it may invoke the wrapper and obtain the contents of \(\ell\) at any time.) The payoff is in the conclusion: the closure returned by readonly \(\ell\) is itself a low value and may therefore be safely shared with untrusted code.

Given this specification for readonly, let us now consider a client of readonly that wishes to maintain an invariant on its private (i.e., high-integrity) state:

```latex
usetwo \triangleq \text{let } r = \text{ref} 2 \text{ in }
\text{let } w = \text{readonly} r \text{ in }
\text{let } use = \lambda x. \text{assert} ((!r) = 2) \text{ in }
(use, w)
```

Here, the expression usetwo allocates a reference cell \(r\) containing the value 2, and returns a pair of functions, one of which simply asserts that \(r\) continues to contain 2, and the other of which is the result of readonly \(r\). The reference cell \(r\) is, however, kept private (i.e., hidden in the environments of the closures use and \(w\)).

Our aim is to verify that usetwo can be safely linked with arbitrary untrusted code. To do this, we first prove that it satisfies the spec \(\{ \top \} \text{usetwo} \{ x. \text{lowval} x \}\) which means that we can always run usetwo and, if we do, it returns a low value. It is easy to see intuitively why this spec holds: reference cell \(r\) always contains 2, which is trivially low, so together with the spec for readonly, this is sufficient to ensure that the closures returned by usetwo are low as well.

Next, we appeal to a general meta-theorem—and one of the main technical results of this paper—called robust safety. Robust safety states that, if some user code satisfies a spec like the one given above for usetwo—i.e., a spec whose postcondition stipulates that the resulting value is low-integrity—then we can run that verified user code under an arbitrary adversarial context \(C\) (i.e., any context \(C\) that does not itself contain any assert statements), and we will be assured that the execution of the resulting program will never result in a violation of any of the user code’s internal assertions. Thus, in the particular case of usetwo, we know that the assertion that \(r\) equals 2 will never fail.

Robust safety is a well-known meta-theorem in the security literature (Bengtson et al. 2011; Gordon and Jeffrey 2001), but it has not heretofore been employed in the context of object capability programming. One of the central contributions of this paper is the observation that robust safety is exactly the property a language must satisfy in order to support OCPs. Moreover, as we will
evaluate (rec f x. e) closed

assert true; σ → (e; σ)

assert false; (h, g) → (e; σ)

ref v; (h, g) → (e; σ)

if e = true then (e; σ)

if CAS v1 v2; σ → (e; σ)

T; σ → T; σ’

Fig. 1. Heap language with assertions

demonstrate via a range of interesting examples, the notion of low-integrity values is essential to compositionally specifying an OCP’s contribution toward the robust safety of programs that use it.

The remainder of the paper is structured as follows. First, in §2, we introduce some basics of OCPL, along with our formalization of low values, explain how we verify our motivating readonly example, and state the key metatheorems of adequacy and robust safety. In §§3–5, we present specifications for several more complex and realistic OCPs, along with examples of how to use those specifications to verify representative clients. In §6, we discuss related work. Finally, in §7, we conclude with a discussion of future work.

2 ROBUST SAFETY AND OCPL

In this section, we define the semantics of the programming language we will be reasoning about (§2.1), present the main ideas of OCPL and show how to use it to reason formally about our motivating example (§2.2), and state key metatheorems for OCPL (§2.3).
2.1 A higher-order concurrent heap language with assertions

The programming language we consider in this paper, which we call HLA, is a fairly standard higher-order concurrent imperative language, essentially the same as the one studied by Krebbers et al. (2017a), but with one important extension. We define the language in Fig. 1. The core of the language is a call-by-value $\lambda$-calculus with recursive functions, products, sums, references, and fork-based concurrency. The definitions of free and bound variables and capture-avoiding substitution are standard and omitted. We endow the language with an operational semantics in the style of Felleisen and Hieb (1992). The head reduction judgment $\sigma \vdash e \rightarrow_\ell e'$ describes the single steps of computation that may be taken by individual threads. It means that, starting in state $\sigma$, a thread running expression $e$ may step to expression $e'$ while changing the state to $\sigma'$ and forking off a list of threads running expressions $\overline{e_f}$. Threadpool reduction $T; \sigma \rightarrow T'; \sigma'$ lifts head reduction to lists of concurrently running threads. We leave implicit the choice of operators unop and binop, but note that they reduce according to (partial) evaluation functions and are subject to a semantic constraint; for example, the reduction rule for unary operators

$$\text{unop } v; \sigma \rightarrow_\ell v'; \sigma; e \quad \text{if eval unop } v = v'$$

employs a partial function eval : $Unop \rightarrow Val \rightarrow Val$ (which, for the robust safety theorem presented below, is assumed to send low-integrity values to low-integrity values). As is standard in semantics for languages with mutable references, locations $\ell$ are not part of the “surface” syntax of the language; they may only arise dynamically during execution. Using compare-and-swap (CAS), one can implement locks (and other concurrency primitives). We use the symbol ‘_’ in place of a bound variable to indicate any variable not occurring free in the scope of the binder. In addition to standard derived forms (e.g., $n$-ary products and let expressions), we write

$$\lambda x. e \equiv \text{rec } x. e$$

$$\lambda x_1 \cdots x_n. e \equiv \lambda x_1. \cdots. \lambda x_n. e$$

$$\text{rec } f x_1 \cdots x_n. e \equiv \text{rec } f x_1. \lambda x_2 \cdots x_n. e$$

and we use pattern-matching notation to deconstruct products, rather than a series of fst and snd projections.

The one new feature in HLA, not present in Krebbers et al.’s calculus, is assertion expressions, assert $e$, which serve to specify safety properties that should always hold dynamically. Assertions in HLA operate a little differently from assertions in languages like C or Java, where failed assertions abort execution or raise an exception. Here, $e$ is a boolean expression. If $e$ evaluates to true, then assert $e$ has no effect. But if $e$ evaluates to false, then assert $e$ has the effect of irreversibly setting a “goodness bit” $g$, which is maintained as part of the machine state, to Fail. (The bit is initially set to OK.) This goodness bit simply checks whether any assertion expression has dynamically failed during execution. Ultimately, our robust safety theorem will use this goodness bit to ensure that, for properly verified code, the goodness bit will remain OK throughout execution (and hence all dynamic assert checks must succeed).

---

3OCPs are typically studied in the context of an event-driven programming model. We work in the more general setting of fork-based concurrency because Iris supports it natively and it does not seem to significantly affect reasoning about OCPs.

4The expression $\text{CAS } v_1 \downarrow v_2$ atomically compares the contents of location $\ell$ to value $v_1$ and, if equal, writes $v_2$ to $\ell$ and returns true; otherwise, it returns false. For simplicity, we do not restrict CAS to comparing first-order, “word-sized” values. This is unrealistic, but we use CAS in only realistic ways. (Spinlocks, in particular, are compatible with such a restriction and suffice for our purposes.)

5We define C-style assertions—which we call assumption expressions—in §3.
The reader may wonder why we employ a goodness bit instead of just saying that assert false gets stuck or aborts the program. The reason is simple: we wish to use OCPL to reason not only about fully verified code but also about the behavior of verified code when linked with untrusted code (that does not itself contain assertions). So it is important that we have a way of verifying Hoare triples even for code that may very well get stuck (i.e., fail to make progress) thanks to dynamic type errors introduced by the untrusted code. We nevertheless want to say that assert expressions in such code always succeed, so we must differentiate assert false from other stuck states.\(^6\)

### 2.2 A program logic for reasoning about OCPs

We now present some of the essential features of OCPL, our logic for reasoning about OCPs.

**Progressive vs. non-progressive triples.** OCPL is a Hoare-style program logic derived from Iris, a modern separation logic for higher-order concurrent imperative programs.\(^7\) As such, one of the main assertions in OCPL is the *Hoare triple*, which is used to specify the behavior of expressions in terms of preconditions and postconditions. The Hoare triple \(\{P\} e [x. Q]\) asserts that, assuming \(e\) is executed starting in a state satisfying the precondition \(P\), then it will execute without any dynamically failing assert expressions, and if it terminates with value \(v\), then the final state will satisfy the postcondition \(Q\) (which may mention the bound variable \(x\)). The *progress bit* \(p \in \{\text{progress, nonprogress}\}\) indicates whether the triple is *progressive* (i.e., ensures that \(e\) does not get stuck) or *non-progressive*.\(^8\) As shorthand, we will write \(\{P\} e \{x. Q\}\) and \(\{P\} e \{x. Q\}\) for progressive and non-progressive triples, respectively; the former assertion implies the latter. We often write the postcondition of a Hoare triple as \(\{x_1 \cdots x_n. ret \text{ pat. } Q\}\), which binds the \(x_i\) and specifies that any returned value will be precisely \(\text{pat}\). This notation is (roughly) syntactic sugar for the post-condition \(\{x. \forall x_1, \ldots, x_n. x = \text{pat} * Q\}\).

Hoare triples in Iris (and thus OCPL) enjoy the standard *structural rules* for higher-order concurrent separation logic (e.g., the rule of consequence, the frame rule, and monadic bind and return) as well as rules for defining and enforcing logical *protocols* on physical and ghost state. Readers interested in these aspects of OCPL should consult Jung et al. (2015) for a high-level overview and Jung et al. (2017) for a recent account of Iris’s foundations.

In addition, OCPL enjoys a full set of *basic rules* for HLA, comprising progressive triples for reasoning about head reductions and non-progressive triples for reasoning about stuck expressions. The following three rules, for example, concern application.

\[
\begin{align*}
\text{App} & \quad (\text{rec } f \; x. \; e) \; \text{closed} \\
\{P\} \; e[v/x, \text{rec } f \; x. \; e/f] \; \{x. \; Q\} & \quad \text{AppNRec} \\
\vdash \{P\} \; (\text{rec } f \; x. \; e) \; v \; \{x. \; Q\} & \quad \text{AppOpen}
\end{align*}
\]

Rule App says that, to prove a triple for a closed function application, it suffices to show an analogous triple after \(\beta\)-reduction. Rules AppNRec and AppOpen give non-progressive triples to applications that, according to the operational semantics of HLA, always get stuck: as such applications never produce a value, they can be given an arbitrary postcondition \(Q\).

---

\(^6\)Instrumenting the state with a goodness bit is not strictly necessary; for example, one could alternatively use a special machine configuration to indicate failed assertions.

\(^7\)Readers unfamiliar with separation logic can, to a first approximation, read separating conjunction \(P \land Q\) as conjunction \(P \land Q\) and magic wand \(P \Rightarrow Q\) as implication \(P \Rightarrow Q\).

\(^8\)Note: this is a point of difference from the original Iris, which only supported progressive triples.
We omit analogous rules for other expressions that ignore the ambient machine state, turning now to the more interesting rules for assertions and locations.

**Assertion expressions.** OCPL supports the following rule for assertion expressions.

\[
\text{Assert} \quad \{ P \} e \{ x. \ x = \text{true} \} \Rightarrow \{ P \} \text{assert} e \{ \text{ret}(). Q \}
\]

**Assert** enables reasoning about successful assertions. It says that, to prove a Hoare triple for `assert e` with postcondition `Q` and return value `()`, it suffices to prove that `e` evaluates to `true` with postcondition `Q`.

**High vs. low locations.** OCPL divides memory locations (i.e., mutable references) into two types: high-integrity and low-integrity (or just high and low for short). High locations are locations that are private to user code, on which it may place invariants of its choosing, and to which untrusted code should not be given direct, unfettered access. Low locations are locations that may be freely shared with—and may in fact have been allocated by—untrusted code. Note that there is no distinction between high and low locations in the operational semantics of HLA; rather, this distinction is merely something we track in OCPL in order to formally specify the interface between user code and untrusted code.

**Reasoning about high locations.** Since high locations are locations controlled privately by verified user code, OCPL supports reasoning about them in essentially the same way as one reasons about pointers in traditional separation logic—via the classic “points-to” assertion \( \ell \leftarrow v \). This points-to assertion denotes the knowledge that location \( \ell \) currently points to value \( v \) as well as ownership of \( \ell \). What this means is that if we own \( \ell \), then no other party (e.g., neither another thread nor the current thread’s context) can be simultaneously making any assertion about \( \ell \), so we should have permission to freely read from and write to \( \ell \) without worrying about violating any other party’s reasoning. This reasoning is formalized with standard rules about high locations displayed in Fig. 2a.

**Reasoning about low locations.** We represent the knowledge that a location \( \ell \) is low via the assertion \( \text{lowloc} \ell \). In contrast to the points-to assertion for high locations, the assertion \( \text{lowloc} \ell \) does not denote any ownership of \( \ell \) because as soon as a location is considered safe to be shared with untrusted code, there is no way of knowing what that code will do with it. Rather, \( \text{lowloc} \ell \) merely tells us that \( \ell \) is low now and will remain low forever. Moreover, if \( \ell \) is low—and hence
safely shareable with untrusted code—then we know that the value \( v \) it points to must also be safely shareable with untrusted code—i.e., \( v \) is a low value.

Of course, this begs the question: what is a low value? Intuitively, a low value is a value from which there is no way to extract a high location. We will make this more precise in a moment. For now, assume we have a predicate lowval, which says that \( v \) is a low value.

The description of low locations given above is formalized in the proof rules of Fig. 2b. Rule \text{HighNotLow} \ says that a location cannot be high and low at the same time. Rules \text{AllocLow}, \text{LoadLow}, and \text{StoreLow} mimic the corresponding rules for allocation, reading, and writing of high locations, except that for low locations, we do not track the precise contents of the location \( \ell \)—we merely insist that \( \ell \) always points to a low value.

Lifting low locations to low values. We return now to the question of what it means for a value \( v \) to be low. Intuitively, a low value is one from which the language constructs of HLA provide no way to get direct access to any high location. This is fundamentally an extensional property, i.e., a property about the observations that a program can make (i.e., the information it can extract) when passed the value \( v \). As such, for those readers familiar with classic techniques from program semantics, it will not come as a surprise that a natural way of formally accounting for this property is via a logical relation.

Fig. 3 shows the definition of this logical relation, presented in a somewhat more general form, lift \( \Psi \ v \). Essentially, lift takes as input \( \Psi \), a predicate on locations, and lifts it to a predicate on values, with the property that lift \( \Psi \ v \) is true if \( \Psi \) holds for any location that can be extracted from \( v \). Given this definition, lowval \( v \) can be defined simply as lift lowloc \( v \) —i.e., a value is low if any location that can be extracted from it is low.

On literals and unit, lift \( \Psi \) is trivially true since no location can be extracted from them; on locations, lift \( \Psi \) is simply \( \Psi \); and on products and sums, lift \( \Psi \) is defined in the obvious recursive manner. The only interesting case is the one for functions. For a function value \( f \), we want to say that lift \( \Psi f \) if, whenever we apply \( f \), the resulting term only ever produces values that satisfy lift \( \Psi \). (This property on applications of \( f \) is expressed via a non-progressive Hoare triple since \( f \) may be applied to arguments constructed by untrusted code.) But what arguments should we consider when applying \( f \)? This is where the logical relation comes in: we need only consider argument values that themselves satisfy lift \( \Psi \), because ultimately we will make sure that all values passed to untrusted code satisfy lift \( \Psi \).

Note that the definition of lift \( \Psi \) in the function case is rather circular, since it quantifies over values that satisfy lift \( \Psi \). To ensure that this definition is well-founded, we rely on OCPL’s support

\[
\text{lift } \Psi (\text{rec } f \ x \ e ) \triangleq \forall v. \ (\text{lift } \Psi v ) e[v/x, \text{rec } f \ x \ e / f ] \{ y. \ \text{lift } \Psi y \},
\]

\[
\text{lift } \Psi \ell \triangleq \Psi \ell
\]

\[
\text{lift } \Psi \text{lit } \triangleq \top
\]

\[
\text{lift } \Psi () \triangleq \top
\]

\[
\text{lift } \Psi (v_1, v_2) \triangleq (\text{lift } \Psi v_1 \ast \text{lift } \Psi v_2)
\]

\[
\text{lift } \Psi (\text{inl } v) \triangleq \text{lift } \Psi v
\]

\[
\text{lift } \Psi (\text{inr } v) \triangleq \text{lift } \Psi v
\]

Fig. 3. Lifting location predicates to value predicates
for guarded recursive definitions (inherited from Iris). In OCPL, a definition may be arbitrarily recursive so long as any recursive references are guarded by an occurrence of the \( \triangleright \) (“later”) modality. Under the hood, all assertions in OCPL are implicitly indexed by steps of computation—i.e., the model of OCPL is a kind of step-indexed model (Birkedal et al. 2011)—and \( \triangleright P \) means that \( P \) holds, roughly speaking, “after one step of computation”. Since the fine points of the “later” modality are not particularly relevant to our main contribution, and our use of “later” is standard, we refer the reader to prior work for a more detailed presentation (Appel et al. 2007).

Using the aforementioned structural and basic rules, we can readily derive rules for working with lifted values. For example, the derived rule

\[
\text{LIFTAPP} \\
\{ \text{lift } \Psi \, v \ast \text{lift } \Psi \, v' \} \, u \, v' \{ x. \text{ lift } \Psi \, x \}
\]

says that one can always apply a lifted value to a lifted argument, and such applications either get stuck (the rule uses a non-progressive triple) or produce a lifted result. These derived rules help, for instance, in verifying clients of the public membrane (§5), which we specify using reading values.

To make concrete this rather abstract tour of OCPL, we return to our motivating example. To verify that \textit{readonly} satisfies its specification, we have to show that the function it returns, \( \lambda \ell \cdot ! \ell \), is a low value given a triple \( \{ \top \} \, ! \ell \, \{ x. \text{ lowval } x \} \) for dereferencing \( \ell \). This follows easily by \texttt{LIFTREC}.

To verify that the expression \texttt{usetwo} returns a low value, we proceed as follows. From the allocation \texttt{ref 2} we obtain \( \ell \leftarrow 2 \) for some location \( \ell \) (by \texttt{ALLOC}). Since we own \( \ell \), we are free to transfer ownership of \( \ell \) into a shared invariant, which in Iris (and OCPL) is written \( \ell \leftarrow 2 \), stipulating that \( \ell \) will always contain 2. (Invariants like \( \ell \leftarrow 2 \), which are established “dynamically” in the course of a program proof and hold persistently thereafter, are a feature inherited directly from Iris (Jung et al. 2015). We omit rules for working with them here since they are orthogonal to the focus of this paper.) By \texttt{LOAD}, we easily obtain a lemma

\[
\ell \leftarrow 2 \vdash \{ \top \} \! \ell \{ \text{ret } 2. \top \} \tag{\dagger}
\]

stating that, under our invariant, dereferencing \( \ell \) always returns 2. To prove that the value \( w \) returned by \texttt{readonly} \( \ell \) is low-integrity, we apply \texttt{readonly}’s specification and have to show that dereferencing \( \ell \) returns a low value (which is trivial, by (\dagger) and \texttt{LIFTLIT}). To prove that the function \texttt{use} \( \equiv \lambda \ell. \text{ assert } (((! \ell) = 2)) \) is low-integrity, we apply \texttt{LIFTREC}. In this subproof, we have to prove that, under our invariant, the assertion returns a low value (trivial by \texttt{ASSERT}, (\dagger), \texttt{LIFTUNIT}). It remains to show that the pair \((w, \text{use})\) is low-integrity given knowledge that both \( w \) and \texttt{use} are low-integrity, which is trivial by \texttt{LIFTPAIR}.

2.3 Metatheory

OCPL inherits from Iris the basic metatheorem of adequacy for progressive triples (omitted here), which asserts that expressions verified with progressive triples make progress (i.e., do not get stuck) and that, if such expressions terminate, the postconditions of their triples actually hold. OCPL also enjoys an analogous metatheorem for non-progressive triples, which simply drops the progress guarantee. More interestingly, OCPL enjoys the following metatheorems concerning dynamically checked safety properties. The entire metatheory has been verified formally in Coq (OCPL 2017).

**Theorem AdequacySafety.** If an expression is verified and run in a good state, we can observe that every reachable state is good (i.e., no assertions fail):

\[
\{ \top \} \, \ell \{ x. \top \}_p \quad \langle e ; (h, \text{OK}) \rangle \to^* \langle T' ; (h', g') \rangle \quad g' = \text{OK}
\]
intervals = \lambda . \text{let } (\text{seal, unseal}) = \text{makeseal}() \text{ in} \nn \text{let } \text{makeint} = \lambda n_1 n_2. \text{seal}(\text{if } n_1 \leq n_2 \text{ then } (n_1, n_2) \text{ else } (n_2, n_1)) \text{ in} \nn \text{let } \text{imin} = \lambda i. \text{fst}(\text{unseal } i) \text{ in} \nn \text{let } \text{imax} = \lambda i. \text{snd}(\text{unseal } i) \text{ in} \nn \text{let } \text{isum} = \lambda i. \text{let } x = \text{unseal } i \text{ in} \lambda j. \text{let } y = \text{unseal } j \text{ in} \nn \text{seal(fst } x + \text{fst } y, \text{snd } x + \text{snd } y) \nn \text{in } (\text{makeint, imin, imax, isum}) \nn \text{Fig. 4. Library for integer intervals}

**Theorem RobustSafety.** Let AdvCtx denote the set of HLA contexts containing neither locations nor assertions. If expression \( e \) is closed and has been verified to return only low values, then for every adversarial context \( C \), on running \( C[e] \) from an initial state, we can observe that every reachable state is good (i.e., no assertions fail):

\[
C \in \text{AdvCtx} \quad e \text{ closed} \quad \{\top\} e \{x. \text{lowval } x\}_p \quad (C[e];(\emptyset, 0K)) \to^* T' ;(h', g') \quad g' = 0K
\]

Robust safety captures our informal distinction between “user code” and “untrusted code”. Both are written in HLA, but user code must be verified in OCPL and may contain assertion statements, whereas untrusted code need not be verified and is not permitted to contain assertion statements. Untrusted code may also not contain references to memory locations \( \ell \), since those are not part of the surface syntax of the language. The theorem says that verified user code can be linked with untrusted code, while remaining safe (i.e., its internal assertion statements will never assert false).

### 3 Dynamic Sealing

We now consider one of the oldest and most influential OCPs: *dynamic sealing*, also called the sealer-unsealer pattern. Originally proposed by Morris (1973), dynamic sealing makes it possible to support data abstraction in the absence of static typing. In this section, we show how OCPL supports compositional reasoning about dynamic sealing. In particular, we show how to implement dynamic sealing in HLA and how to give a compositional specification for this implementation, from which we derive useful specifications for interesting abstractions built on top of it and prove robust safety for representative clients. (We consider an alternative implementation of the OCP, satisfying a slightly weaker specification, in Appendix A. We also show how ideal specifications for cryptographic signing and encryption primitives can be derived from the specification of dynamic sealing in Appendix B.)

The functionality of dynamic sealing. Morris (1973) introduced dynamic sealing to enforce data abstraction while interoperating with untrusted, potentially ill-typed code. He stipulated a function makeseal for generating pairs of functions (seal, unseal), such that (i) for every value \( v \), seal \( v \) returns a value \( v' \) serving as an opaque, low-integrity proxy for \( v \); and (ii) for every value \( v' \), unseal \( v' \) returns \( v \), if \( v' \) was produced by seal \( v \), and otherwise gets stuck. The key point is that this seal-unseal pair supports data abstraction: the client of these functions can freely pass sealed values to untrusted code since they are low-integrity, while at the same time imposing whatever internal invariant it wants on the underlying values that they represent.
To see how this is useful, consider a simple client of dynamic sealing, namely Morris’s example of a library for integer intervals (Fig. 4). Let \([n_1, n_2]\) denote the set \(\{n_1, n_1 + 1, \ldots, n_2 - 1, n_2\}\).

The function \textit{intervals} returns several interval-manipulating routines: \textit{makeint} \(n_1 n_2\) constructs \([\min n_1 n_2, \max n_1 n_2]\). \textit{imin} \(n_1 n_2\) returns \(n_1\), \textit{imax} \(n_1 n_2\) returns \(n_2\), and \textit{isum} \(n_1 n_2\) returns \([n_1', n_2']\).

These routines use dynamic sealing to enforce the internal data representation of intervals, namely that the interval \([n_1, n_2]\) is represented by the pair \((n_1, n_2)\), which is critical to ensuring correctness of the library. In particular, notice that \textit{seal} and \textit{unseal} are kept private to the \textit{intervals} implementation, which means it can enforce that the only values sealed with \textit{seal} are pairs \((n_1, n_2)\) representing intervals (i.e., where \(n_1 \leq n_2\)). Consequently, the \textit{imin} (resp. \textit{imax}) function can simply return the first (resp. second) component of its argument after unsealing it, because it \textit{k}nows that, even if the argument comes from untrusted code, \textit{so long as the unsealing succeeds}, the resulting value will be a pair where the first (resp. second) component represents the lower (resp. upper) bound of the input interval. Furthermore, a client of \textit{intervals} knows that if it applies both \textit{imin} and \textit{imax} to an arbitrary untrusted value \(v\), and both operations succeed, producing values \(v_1\) and \(v_2\), then \(v\) must indeed represent a proper interval \([n_1, n_2]\) (with \(n_1 \leq n_2\)), and the resulting values of \textit{imin} and \textit{imax} are precisely \(v\)’s lower and upper bounds (i.e., with \(v_1 = n_1\) and \(v_2 = n_2\)). In this way, dynamic sealing affords programmers the essential functionality of data abstraction, even when interfacing with untrusted code, at the cost of some simple dynamic checks at the boundaries of the abstraction.

\textbf{Implementation of dynamic sealing.} We implement dynamic sealing in HLA as follows.\(^9\)

\[
\text{makeseal} \triangleq \lambda_. \text{let tbl = ref mapempty in} \\
\quad \text{let sync = makesync() in} \\
\quad \text{let seal = } \lambda x. \text{let k = ref () in sync(} \lambda _. \text{tbl }\leftarrow \text{mapinsertnew(!tbl) k x}; k \text{ in} \\
\quad \text{let unseal = } \lambda k. \text{assume (isloc k); sync(} \lambda _. \text{maplookup(!tbl) k in} \\
\quad (\text{seal, unseal})
\]

The function \textit{makeseal} allocates a fresh lookup table \textit{tbl}, which is used to store the mapping between sealed values and their proxies, and access to which is shared by the \textit{seal} and \textit{unseal} functions that \textit{makeseal} returns. To ensure proper synchronization between table accesses during calls to \textit{seal} and \textit{unseal}, \textit{makeseal} also creates a lock via the call to \textit{makesync}. (The latter returns a higher-order function \textit{sync}, which ensures mutual exclusion among any expressions \(e_i\) run under \textit{sync(} \(\lambda_. \ e_i\)).

For more details on \textit{makesync}, see Appendix F.) An entry \([k \mapsto v]\) in \textit{tbl} signifies that location \(k\) is a low-integrity proxy for sealed value \(v\). To seal a value \(v\), we allocate a fresh, low location \(k\), extend the table, and return \(k\). To unseal a value \(v'\), we require that \(v'\) be a location \(k\) and then look up \(k\) in the table.

To test whether the argument to \textit{unseal} is a location, we use a primitive unary operator \textit{isloc} (with boolean return type), as well as an \textit{assumption expression}. The expression \textit{assume} \(e\) resembles a C- or Java-style assertion: it returns unit if \(e\) evaluates to \textit{true}; otherwise, it gets stuck. It is easy to define assumption expressions using a stuck term \textit{abort}:

\[
\text{abort } \triangleq 00 \quad \text{assume } e \triangleq \text{if } e \text{ then } () \text{ else abort}
\]

\(^9\)We discuss the straightforward functions \textit{makesync}, \textit{mapempty}, \textit{mapinsertnew}, and \textit{maplookup} in Appendix F. Note that \textit{maplookup f k} gets stuck if location \(k\) is not in the domain of the partial function represented by value \(f\).
Data abstraction.

\[\text{MAKESealSpec} \{ \top \} \text{makeseal}() \{ v_1, v_2, y, \text{ret}(v_1, v_2). \text{isseal} y v_1 \phi \text{ isunseal} y v_2 \phi \}\]

\[\text{SealSpec} \{ \text{isseal} y s \phi \phi v \} s v \{ x'. \text{issealed} y v x' \phi \}\]

\[\text{UnsealSpec} \{ \text{isunseal} y u \phi \phi v \} u v' \{ \text{ret} v. \top \} \]

\[\text{UnsealAnySpec} \{ \text{isunseal} y u \phi \text{ issealed} y x v' \phi \}\]

\[\text{SealedInv} \text{issealed} y v v' \phi \phi v' \]

\[\text{SealedAgree} \text{issealed} y v v' \phi \text{ issealed} y v_2 v' \phi \phi v_1 = v_2\]

\[\text{sealedLow} \text{issealed} y v v' \phi \text{ lowval} v' \]

\[\text{UnsealedLow} (\forall v) \text{lowval} v \phi \phi v \]

\[\text{Assume} \{ P \} e \{ x. x = \text{true} \rightarrow Q \}, \{ P \} \text{assume} e \{ \text{ret} () . Q \},\]

\[\text{Abort} \{ \top \} \text{abort} \{ x. Q \},\]

These satisfy the following specifications.

\[\text{Abort}\]

\[\text{Assume}\]

\[\text{SealLow} (\forall v) \text{lowval} v \phi \phi v \]

\[\text{UnsealedLow} (\forall v) \phi \phi v \text{ lowval} u \]

\textit{Specifying dynamic sealing.} The motivation for dynamic sealing given above was completely informal. Let us now see how OCPL lets us formalize it.

We specify dynamic sealing in Fig. 5. Rule \text{MAKESealSpec} for allocating a sealer-unsealer pair ties the two functions it returns to an abstract (logical) name \(y\) and representation invariant \(\phi\). In particular, the assertions\(^{10}\) \text{isseal} \(v_1 \phi\) and \text{isunseal} \(v_2 \phi\) represent knowledge that \(v_1\) and \(v_2\) are the seal and unseal functions associated with the sealer-unsealer pair named \(y\) with representation invariant \(\phi\). Similarly, the assertion \text{issealed} \(y v v' \phi\) represents knowledge that \(v'\) is a low-integrity proxy for value \(v\) (for the sealer-unsealer pair named \(y\) with representation invariant \(\phi\)). This relation is functional in \(v'\)—i.e., unsealing the same \(v'\) twice will produce the same \(v\) (\text{SealedAgree})—and implies \(\phi \phi v\)—i.e., that the unsealed value \(v\) must satisfy the representation invariant (\text{SealedInv}). Conversely, to seal a value \(v\), one is required to prove \(v v\) (\text{SealSpec}). The progressive triple \text{UnsealedSpec} is suitable for use when one already knows that \(v\) and \(v'\) are related; it returns \(v\). The non-progressive triple \text{UnsealAnySpec} returns the value \(v\) related to \(v'\), or gets stuck.

Finally, the three rules at the bottom of Fig. 5 concern the lowval relation. Rule \text{SealedLow} says that all sealed values are low values—indeed, the whole point of using sealing is to produce low-integrity proxy values—while rules \text{SealLow} and \text{UnsealedLow} say when the seal and unseal functions themselves

\(^{10}\)In our specifications, predicates whose names start with "is" produce, when fully applied, \textit{persistent} propositions (and are themselves called persistent predicates). Persistent propositions may be regarded as intuitionistic (technically, they may be freely duplicated). A sealer-unsealer pair’s representation invariant \(\phi\) must be persistent.
Properties.

\[
\text{MinSpec} \quad \{ \text{ismin } y \ \text{imin } * \ \text{isinterval } y \ n_1 \ n_2 \ i \} \ \text{imin } i \ \{ \text{ret } n_1. \ \top \}
\]

\[
\text{IntervalInv} \quad \text{isinterval } y \ n_1 \ n_2 \ i \vdash n_1 \leq n_2 \\
\text{IntervalAgree} \quad \text{isinterval } y \ n_1 \ n_2 \ i \ast \text{isinterval } y \ n'_1 \ n'_2 \ i \vdash n_1 = n'_1 \ast n_2 = n'_2
\]

Non-progressive triples.

\[
\text{MinAnySpec} \quad \{ \text{ismin } y \ \text{imin }\} \ \text{imin } v \ \{ n_1 \ n_2, \ \text{ret } n_1. \ \text{isinterval } y \ n_1 \ n_2 \ v \}?
\]

\[
\text{MaxAnySpec} \quad \{ \text{ismax } y \ \text{imax }\} \ \text{imax } v \ \{ n_1 \ n_2, \ \text{ret } n_2. \ \text{isinterval } y \ n_1 \ n_2 \ v \}?
\]

Low values.

\[
\text{IntervalLow} \quad \text{isinterval } y \ n_1 \ n_2 \ i \vdash \text{lowval } i \\
\text{MinLow} \quad \text{ismin } y \ \text{imin } \vdash \text{lowval } \text{imin} \\
\text{MaxLow} \quad \text{ismax } y \ \text{imax } \vdash \text{lowval } \text{imax}
\]

Fig. 6. Intervals interface (selected rules—see Appendix C)

can be considered low values. Intuitively, seal and unseal can be considered low values whenever doing so does not either (i) violate the representation invariant \(\phi\) or (ii) result in high-integrity values flowing to a low context. Thus, seal is low so long as any value that is passed to it (i.e., any low value) satisfies \(\phi\), and unseal is low so long as any value returned by it (i.e., any value that satisfies \(\phi\)) is low. (We show how to gainfully employ low seal and unseal functions in Appendix B.)

The specification for dynamic sealing from Fig. 5 can be straightforwardly used to derive an analogous specification for the interval-manipulation routines generated by intervals, as we described above. In particular, by choosing the representation invariant \(\phi \ v \triangleq \exists n_1, n_2. \ v = (n_1, n_2) \ast n_1 \leq n_2\), we can easily verify the rules shown in Fig. 6 (the full specification for intervals is given in Appendix C). The assertion isinterval \(y \ n_1 \ n_2 \ i\) represents knowledge that value \(i\) represents interval \([n_1, n_2]\) in the instance of the intervals library with abstract name \(y\). A value may represent at most one interval (IntervalAgree). A set of progressive triples (e.g., MinSpec) account for our informal specification and are suited to reasoning about values known to represent intervals. A set of non-progressive triples are suited to reasoning about values that might represent intervals. Rule MinAnySpec, for example, says that for any value \(v\), imin \(v\) might get stuck, but if it terminates with value \(\hat{v}\), then \(v\) represents some interval with minimum \(n_1\) and \(\hat{v} = n_1\). Lastly, there are rules (e.g., IntervalLow and MinLow) expressing the fact that values representing intervals, and the interval routines themselves, may be safely shared with untrusted code. Crucial to the soundness of these rules is the fact that the seal function used internally by intervals is not shared with untrusted code (i.e., not low): according to rule SealLow, in order for seal to be treated as low, the representation invariant \(\phi\) would have to be satisfied by all low values, which it clearly is not.

With the spec for intervals in hand, we can readily prove that the following simple but illustrative client of intervals is robustly safe.

\[
\text{client} \triangleq \lambda \text{cap} = \text{intervals}() \in \\
\text{let } (\text{makeint, imin, imax, isum}) = \text{cap} \in \\
\text{let } \text{check} = \lambda j. \text{assert} (\text{imin } j \leq \text{imax } j) \in \\
\text{(check, cap)}
\]
The expression constructs an instance $cap$ of the intervals library, along with a function $check$ which takes an arbitrary value $j$ as its argument and asserts that $imin\ j$ is no greater than $imax\ j$. Intuitively, even if $client$ is shared with untrusted code, this assertion must always succeed, because if the applications of $imin$ and $imax$ do not get stuck, it means that $j$ is a proper interval value, whose lower bound is $\leq$ its upper bound.

Formally, this is guaranteed as follows. First, using the spec for $intervals$, we can prove $\{\top\} \ client \ \{x.\ lowval\ x\}$

The key step in the proof involves showing that the assertion in $check$ succeeds for arbitrary $j$, which follows directly from $MinAnySpec, MaxAnySpec, IntervalAgree$, and $IntervalInv$. Second, we appeal to Theorem $RobustSafety$ from §2.3, which implies that the assertion in $check$ will not fail, even when $client$ is linked with untrusted code.

4 CARETAKER

Next, we consider another well-known OCP, the caretaker pattern (Miller and Shapiro 2003; Miller 2006). This OCP allows verified (trustworthy) code to grant untrusted code access to a high-integrity resource (a high-integrity location or an API that modifies high-integrity locations), and subsequently disable or enable the access at any time. When the caretaker is enabled, untrusted code can access the resource; when the caretaker is disabled, untrusted code cannot access the resource and the verified code has full control over the resource. The caretaker pattern is useful when the verified code wants to ensure that the untrusted code can access the resource only while some invariant holds. Disabling the caretaker allows the verified code to temporarily break the invariant, secure in the knowledge that untrusted code won’t be able to access the resource until the caretaker is re-enabled.

In this section, we first implement a caretaker for APIs in HLA, and specify and verify it in OCPL. Then, we use the API caretaker to implement a second caretaker for locations and, again, specify and verify it. Finally, we present a simple client for the location caretaker and establish that it is robustly safe.

API caretaker. The API caretaker allows verified code to share revokable access to any set of functions, which may have side-effects.

$$makecaretaker \triangleq \lambda_\_.\ \text{let enabled = ref false in let sync = makesync() in (sync, enabled)}$$

$$\quad \text{wrap} \triangleq \lambda(\text{sync, enabled})\ f\ x.\ \text{sync}(\lambda_.\ \text{assume(!enabled); f x})$$

$$\quad enable \triangleq \lambda(\text{sync, enabled}).\ \text{sync}(\lambda_.\ \text{enabled} \leftarrow \text{true})$$

$$\quad disable \triangleq \lambda(\text{sync, enabled}).\ \text{sync}(\lambda_.\ \text{enabled} \leftarrow \text{false})$$

The function $makecaretaker$ returns a new caretaker $ct$, which comprises a fresh lock $sync$, whose purpose is described soon, and a fresh boolean reference $enabled$. The caretaker is disabled (enabled) when $enabled$ is $\text{false (true)}$. The functions $enable\ ct$ and $disable\ ct$ enable and disable the caretaker $ct$ by setting $enabled$ appropriately. The function $wrap\ ct\ f$ wraps the function $f$ in the caretaker $ct$, returning a function that behaves exactly like $f$ when $ct$ is enabled, and gets stuck when $ct$ is disabled. Additionally, accesses to all functions wrapped in $ct$ are serialized using the lock $sync$.

To use this interface, verified code creates a caretaker $ct$ and holds it privately. It can then wrap any number of API functions using $wrap$ and disclose the wrapped functions to untrusted code. The untrusted code’s access to all those functions can be simultaneously disabled and enabled by calling $disable\ ct$ and $enable\ ct$, respectively.
API caretaker specification. The goal of using the API caretaker is to ensure that untrusted code can access wrapped functions only while some logical resource or invariant $R$ holds on the heap. Our specification of the API caretaker in Fig. 7 formalizes this intuition. The assertion iscaretaker $\gamma$ ct $R$ represents knowledge that value $ct$ is a caretaker with (logical) name $\gamma$ whose wrappers allow (untrusted code) access to wrapped functions only when $R$ holds. The assertion enabled $\gamma$ $b$ represents exclusive ownership of the (verified code’s) right to enable and disable all wrappers created using the caretaker named $\gamma$ as well as knowledge that the caretaker is currently enabled (if $b = true$) or disabled (otherwise).

When applying MakeCaretakerSpec, one may pick an arbitrary invariant $R$. The spec returns a disabled caretaker tied to $R$ and a fresh name $\gamma$. To enable $ct$, the (verified) code calling enable must establish that $R$ holds (rule EnableSpec). Dually, rule DisableSpec says that disabling $ct$ provides ownership of $R$ to the caller (verified code).

Rule WrapSpec is at the heart of our specification. It says that to wrap a function $f$ using $ct$, one has to prove that $f$, when applied to any low value $v$ (possibly provided by the untrusted code), preserves the invariant $R$ and returns a low value. We capture this client responsibility in the precondition (canwrap $f$ $R$) on wrap. So, any wrapped function $f$ can assume $R$ in its precondition but it must re-establish $R$ if it terminates (note that the triple for $f$ $v$ in rule CanWrap is non-progressive). This ensures that while $ct$ is enabled, calls to the wrapped functions by the untrusted code preserve $R$. While $ct$ is disabled, verified code can change the state to break $R$ but, prior to re-enabling $ct$, it must re-establish $R$.

The implementation of the API caretaker verifies against this specification. We note that the specification and verification of the API caretaker closely resemble similar efforts for specifying and verifying locks in concurrent programs (Dinsdale-Young et al. 2010). Based on this observation, we offer a shorter implementation of the API caretaker in Appendix D. This implementation uses lock release and acquire instead of the boolean reference enabled to enable and disable the caretaker.

The API caretaker is a very general OCP. It can be used to build caretakers for other kinds of resources. As an instance, we show next how to build a caretaker for locations using the API caretaker.
enabled γ b₁ * enabled γ b₂ ⊢ ⊥

\( \text{IsRmon} \) \( \text{isrmon } f \Psi \triangleq \forall v. \{\Psi v\} f v \{x'. \Psi v \ast \text{lowval } x'\} \)

\( \text{ISWMon} \)

\( \text{ISWmon } f \Psi \triangleq \forall v. \{\text{lowval } v\} f v \{x'. \Psi x'\} \)

\( \text{MakeLocCtSpec} \)

\{isrmon \( f_\ell \Psi \ast \text{iswmon } f_\ell \Psi \}\}

\( \text{makelocct } f_\ell f_\ell \ell \)

\{ct γ v, ret (ct, v). islocct γ \( ct \ell \Psi \ast \text{enabled } γ \false \ast \text{lowval } v\}\}

\( \text{LocCtEnableSpec} \)

\{islocct γ \( ct \ell \Psi \ast \text{enabled } γ \false \ast \ell \rightharpoonup v \ast \Psi v\} \text{ enable ct } \{\text{ret }(). \text{ enabled } γ \true\}\)

\( \text{LocCtDisableSpec} \)

\{islocct γ \( ct \ell \Psi \ast \text{enabled } γ \true\}\) \text{ disable ct } \{v, \text{ret }(). \text{ enabled } γ \false \ast \ell \rightharpoonup v \ast \Psi v\}\)

Fig. 8. Location caretaker interface

Location caretaker. The location caretaker, defined by the function \( \text{makelocct} \) below, facilitates revokable and mediated read and write access to a location. Specifically, it ensures that the location is accessed by untrusted code only while some stipulated predicate \( \Psi \) holds on the location’s contents.

\( \text{makelocct } \triangleq \lambda \text{rmon } w\text{mon } r. \) \text{let } ct = \text{makecaretaker }() \text{ in}

\text{let } \text{read } = \text{wrap ct } (\lambda_. \text{rmon } (!r)) \text{ in}

\text{let } \text{write } = \text{wrap ct } (\lambda x. r \leftarrow (w\text{mon } x) \text{ in}

(\text{ct, (read, write)})\)

The function \( \text{makelocct} \), when applied to a function \( f_\ell \), a function \( f_\ell \), and a location \( \ell \), returns a pair \((ct, (\text{read, write})\)). The API caretaker \( ct \) controls whether or not the functions \{\text{read, write}\} are enabled. When \( ct \) is enabled, the function \text{read} reads \( \ell \) and filters the read value through \( f_\ell \). Similarly, \text{write} writes \( \ell \) after filtering the value to be written through \( f_\ell \). Thus, \( f_\ell \) and \( f_\ell \) mediate reads from and writes to \( \ell \). Consider, for example, an application \text{write } v, representing a request to write value \( v \) to location \( \ell \). If \( ct \) is enabled, the application reduces to \( \ell \leftarrow (f_\ell v) \) and \( f_\ell \) gets to decide what is written.

To use the location caretaker on a location \( \ell \), verified code invokes \( \text{makelocct } f_\ell f_\ell \ell \) with appropriate \( f_\ell \) and \( f_\ell \). It holds \( \ell \) and the returned API caretaker \( ct \) private, but passes \text{read} and \text{write} to untrusted code. It can then enable and disable access to the protected location using the calls \text{enable ct} and \text{disable ct}, respectively. While \( ct \) is enabled, all reads and writes to \( \ell \) are mediated by \( f_\ell \) and \( f_\ell \), respectively.

Location caretaker specification. Our location caretaker specification (Fig. 8) ensures that the value in the protected location \( \ell \) satisfies some stipulated predicate \( \Psi \) whenever the location is accessible to the untrusted code. The proposition \( \text{islocct } γ \ell \ell \Psi \) means that value \( ct \) is an API caretaker for location \( \ell \) with logical name \( γ \) and predicate \( \Psi \) governing the contents of location \( \ell \) when the caretaker is enabled. Logical names and propositions of the form enabled \( γ b \) serve the same purpose as in the API caretaker interface.

The triple \( \text{MAKELocCtSpec} \) specifies the function \( \text{makelocct} \) and imposes conditions on the function’s first two arguments, \( f_\ell \) and \( f_\ell \). Since \( f_\ell \) receives a value read from the protected location, \( f_\ell \)’s input can be assumed to satisfy \( \Psi \). Since the output of \( f_\ell \) will be returned to untrusted code, that output
must be low. Hence, \( f_r \) must transform values satisfying \( \Psi \) to low values. Dually, since \( f_w \) receives a value from untrusted code and its output is written to the protected location, \( f_w \) must transform low values to values satisfying \( \Psi \). The postcondition for \( \text{makelocct} \) says that the returned functions \( \text{read}, \text{write} \) are low, so they can be passed to untrusted code safely. The precondition of \( \text{enable} \) insists that \( \ell \) point to a value \( v \) satisfying \( \Psi \) before the caretaker is enabled (rule \( \text{LocCyEnableSpec} \)). Dually, the postcondition of \( \text{disable} \) reveals the fact that \( \ell \) points to a value \( v \) satisfying \( \Psi \) (rule \( \text{LocCyDisableSpec} \)).

The location caretaker’s code satisfies this specification, assuming the specification of the API caretaker (Fig. 7). To illustrate how programs may use a location caretaker, and how such programs may be verified, we consider the following very simple program, \( \text{client} \).

\[
\begin{align*}
\text{asserteven} & \triangleq \lambda n. \text{assert (even } n\text{)}; n \\
\text{assumeeven} & \triangleq \lambda n. \text{assume (even } n\text{)}; n \\
\text{client} & \triangleq \text{let } r = \text{ref 0 in } \\
& \quad \text{let } (ct, w) = \text{makelocct asserteven assumeeven } r \text{ in } \\
& \quad \text{enable } ct; \\
& \quad \text{let } sync = \text{makesync () in } \\
& \quad \text{let } use = \text{sync}(\lambda _. \text{disable } ct; \text{assert even } (!r); \\
& \qquad \quad r \leftarrow 1; r \leftarrow 0; \text{enable } ct) \\
& \qquad \quad \text{in } (use, w)
\end{align*}
\]

The expression \( \text{client} \) constructs a reference cell \( r \) (initially containing 0) and constructs and immediately enables a location caretaker \( ct \) serving access to \( r \). The goal is to maintain the invariant that untrusted code only sees even numbers in \( r \). To ensure this invariant, the location caretaker \( ct \) is created with the write filter \( f_w = \text{assumeeven} \), which gets stuck unless the value being written is an even number. The read monitor, \( f_r = \text{asserteven} \), checks that the value being read from \( r \) is indeed even. Additionally, \( \text{client} \) exposes to its (untrusted) context a function \( \text{use} \) that (within the scope of an exclusive lock, \( \text{sync} \)) locally disables the caretaker and temporarily breaks the invariant by writing 1 to \( r \).

Using the specification of the location caretaker, we prove that \( \text{client} \) satisfies the triple

\[
\{ \top \}\ \text{client} \{ x. \text{lowval } x\}
\]

By Theorem RobustSafety, \( \text{client} \) is robustly safe. This means that irrespective of how the (untrusted) context calls \( \text{use} \) and the wrapped read/write interface \( w \), it can never observe a non-even number in \( r \) (else, the assertion in \( \text{asserteven} \) would fail).

5 MEMBRANE

When verified (trustworthy) and untrusted code interoperate, a general concern is that values passed from verified to untrusted code may accidentally reveal high-integrity locations. The membrane pattern (Miller 2006; Miller et al. 2008; Google, Inc. 2015) was designed to offset this possibility. A membrane is a bidirectional transformation on values passing from verified to untrusted code and vice versa. In the verified-to-untrusted direction, it sanitizes values to make the extraction of high-integrity locations impossible. A trivial way to implement the membrane is to use the \( \text{seal} \) function of \( \S 3 \), which replaces all values with proxies. However, these proxies cannot be \( \text{used} \) by untrusted code, other than by passing them back to the verified code. To address this shortcoming, the membrane pattern uses a \textit{deep inspection} of values to selectively hide high-integrity locations.
inside values, while preserving their overall structure (pairs map to pairs, functions map to functions, etc.). This allows the untrusted code to use the transformed values, while still ensuring that it won’t get access to high-integrity locations.

While the specifics of the transformation depend on the language and security objectives of the application, a common core of all membranes is a function that lifts a use case-specific bidirectional transformation on locations to a bidirectional transformation on values. We first define this function, membrane, in HLA below, show its OCPL specification and verify the function. Then, we use this function to implement a specific membrane that we call the public membrane. This membrane replaces each high-integrity location nested in a value with a fresh shadow location and provides the verified code special functions to inspect those shadow locations. This is similar to how the membrane pattern is implemented in Google’s Caja, a JavaScript library for securing communication between mutually distrusting domains in web applications (Miller et al. 2008; Google, Inc. 2015).11

Membrane code. The function membrane that forms the common core of the membrane pattern is defined below. This is a higher-order function that takes as its first two arguments two other functions, locout and locin, both of type Loc → Val. The function locout defines how locations crossing from verified to untrusted code are transformed by the membrane. Dually, the function locin defines how locations crossing in the other direction, from untrusted to verified code, are transformed. Given these two arguments, membrane locout locin is a function of type Val → Val that transforms values passing from verified to untrusted code. The function for transforming values passing in the other direction—from untrusted to verified code—is obtained by reversing the arguments to membrane, i.e., it is membrane locin locout.

\[ \text{membrane} \triangleq \text{rec memb locout locin}. \]

let wrap = memb locout locin in
if isfun x then let unwrap = memb locin locout in
\( \lambda y. \text{wrap}(x(\text{unwrap}y)) \)
else if isloc x then locout x
else if islit x then x
else if x = () then ()
else if ispair x then (wrap(fst x), wrap(snd x))
else if inl x as inl x’ ⇒ inl(wrap x’)
else if inr x as inr x’ ⇒ inr(wrap x’)
else assert false \hspace{1cm} // i.e., not reached

Internally, membrane recurses on the structure of the value \( x \) being transformed. When the value is a location, the result is simply locout \( x \). When the value is a literal, it is returned immediately. When the value is a pair, membrane recurses on the two components of the pair. The interesting case arises when the value \( x \) is a function. In this case, membrane returns a function, which when applied (by the untrusted code), first recursively applies the membrane to the untrusted argument \( y \), then applies the given function \( x \) to the transformed argument, and then re-applies the membrane recursively to the result. Importantly, the membrane is applied in the untrusted-to-verified direction to the function argument \( y \) and in the verified-to-untrusted direction to the result of the function. Technically, the function \( x \) is transformed to \( \text{wrap} \times \circ \text{unwrap} \), where \( \text{wrap} \triangleq \text{memb locout locin} \) and

11Google’s Caja partly automates the invocation of the membrane transformation by rewriting untrusted code. The automation is orthogonal to the specification and verification of the membrane pattern itself, so we do not model it here.
\( \text{ismon } p \nu \Psi_1 \Psi_2 \triangleq \forall a. \{ \Psi_1 a \} \nu a \{ a', \text{ret } a' \}. \Psi_2 a' \)\)

\[
\text{MembraneSpec} \\
\{ \text{ismon } p \text{ locout } \Psi_1 \Psi_2 \ast \text{ismon } p' \text{ locin } \Psi_2 \Psi_1 \} \\
\text{membrane locout locin} \\
\{ w. \text{ ismon } p \ w (\text{lift } \Psi_1) (\text{lift } \Psi_2) \}
\]

Fig. 9. Membrane interface (presupposing \( \Psi_1, \Psi_2 \) persistent)

unwrap \( \triangleq \text{memb locin locout} \) are recursive instantiations of membrane in the verified-to-untrusted and untrusted-to-verified directions, respectively.

**Membrane specification.** Our specification of membrane formalizes the intuition that membrane lifts transformations on locations to transformations on values. The specification is shown in Fig. 9. The defined predicate ismon \( p \nu \Psi_1 \Psi_2 \) means that \( \nu \) is a function that transforms values satisfying the predicate \( \Psi_1 \) to values satisfying the predicate \( \Psi_2 \) (the progress bit \( p \) is needed for technical reasons, that readers may ignore). The specification of membrane says that if locout transforms locations satisfying \( \Psi_1 \) to values satisfying \( \Psi_2 \) and locin does the reverse, then membrane locout locin transforms values satisfying lift \( \Psi_1 \) to values satisfying lift \( \Psi_2 \), where lift is the predicate transformer defined in Fig. 3. Hence, membrane really “lifts” the transformation on locations to a transformation on values in a precise technical sense.

This specification of membrane is very general, since it holds for any predicates \( \Psi_1 \) and \( \Psi_2 \). In any use of membrane, these predicates can be instantiated to match what the arguments locout and locin do. (Later, we show a specific instantiation with \( \Psi_2 = \text{lowloc} \).)

Despite the specification’s generality, membrane is easily verified against it. In the recursive cases, the proof obligations match the inductive hypotheses precisely because of the concordance between the pattern matches in the definitions of lift (Fig. 3) and membrane.

**Public membrane.** Next, we describe how the function membrane can be used to construct a specific membrane, which we call the public membrane. This membrane is similar to the membrane used in Google’s Caja library (Miller et al. 2008; Google, Inc. 2015). The public membrane maintains a unique low-integrity shadow location for every high-integrity location that the verified code declares as important. When values cross from verified to untrusted code, all nested high-integrity locations in them are effectively “replaced” with the corresponding low-integrity shadows (this transformation is implemented using membrane). Thus, the untrusted code only sees shadow locations, not the high-integrity locations. Additionally, the public membrane provides the verified code special functions, shadowread and shadowwrite, to read and write the contents of the shadow locations.

To understand why this is useful, consider a library that allocates an integer reference \( \ell \), and shares it with (untrusted) clients as an I/O buffer. Clients are expected to write only positive integers to \( \ell \), although the library does not strictly require this and the library’s algorithms can execute safely even if the integer is not positive. Over time, many clients of this library have been written. Now, suppose that the library is updated to use different algorithms that really require \( \ell \) to always be positive (else they crash). The obvious way to do this would be to rewrite the library to hold \( \ell \) private, and to export two closures that read and write \( \ell \), the latter only after checking that the value being written is positive. However, this change breaks compatibility with all existing clients, since they must now be rewritten to invoke the new closures to access \( \ell \). The public membrane offers a general solution to this problem. Rather than export closures, the library can deploy a
When a public membrane’s table sends location with corresponding private locations. At any point, the verified code can access shadow locations of private locations by consulting the table. Both functions get stuck on locations not in the table. The functions \( \text{pubwrap} \) and \( \text{pubin} \) transform between private and shadow locations by consulting the table. Both functions get stuck on locations not in the table. The functions \( \text{pubwrap} \) and \( \text{pubin} \) lift these to transformations between values by applying the function \( \text{membrane} \). Effectively, \( \text{pubwrap} \) replaces all private locations in the value passed to it with their shadow locations and \( \text{pubunwrap} \) does the opposite.

The expression \( \text{multipub}() \) creates a public membrane (consistently denoted \( m \)), which comprises a pair \(( \text{sync, tbl} )\) where the reference cell \( \text{tbl} \) contains a finite partial bijection on locations protected by (the lock buried in) \( \text{sync} \). When a public membrane’s table sends location \( \ell_1 \) to \( \ell_2 \), we say that location \( \ell_1 \) is a (high-integrity) private location and location \( \ell_2 \) its (low-integrity) shadow location. The functions \( \text{pubout} \) and \( \text{pubin} \) transform between private and shadow locations by consulting the table. Both functions get stuck on locations not in the table. The functions \( \text{pubwrap} \) and \( \text{pubunwrap} \) lift these to transformations between values by applying the function \( \text{membrane} \). Effectively, \( \text{pubwrap} \) replaces all private locations in the value passed to it with their shadow locations and \( \text{pubunwrap} \) does the opposite.

The expression \( \text{pubref} m v_1 \) constructs a new private location initially containing value \( v_1 \). In addition, it allocates a shadow location (called \( r_2 \) in the code) that initially contains the low-integrity counterpart of \( v_1 \). It stores an association between the two locations in \( m \)’s table. The shadow location \( r_2 \) can be retrieved from the table by applying either \( \text{pubout} \) or \( \text{pubwrap} \) to the private location returned by \( \text{pubref} \).

The function \( \text{shadowread} \) inspects a private location’s shadow location, converting its low-integrity contents into a high-integrity value using \( \text{pubunwrap} \), whereas \( \text{shadowwrite} \) updates a private location’s shadow location after converting the given value to a low-integrity value using \( \text{pubwrap} \).

To use this public membrane interface, the verified code first creates a new public membrane using \( \text{makepub} \). Subsequently, it allocates private locations that might flow to untrusted code using the function \( \text{pubref} \) instead of the language construct \( \text{ref} \). Before sending any value to untrusted code, it invokes \( \text{pubwrap} \) to replace all nested private locations with their shadows. Dually, after receiving any value from untrusted code, it applies \( \text{pubunwrap} \) to replace nested shadow locations with corresponding private locations. At any point, the verified code can access shadow locations.

\[ \text{makepub} \triangleq \lambda \_, \_ \text{let tbl} = \text{ref bijempty in} \]
\[ \text{let sync} = \text{makesync in} \]
\[ \text{(sync, tbl)} \]
\[ \text{pubout} \triangleq \lambda (\text{sync, tbl}) r_1. \]
\[ \text{sync}(\_, \_, \text{bijlookup}(\text{!tbl}) r_1) \]
\[ \text{pubin} \triangleq \lambda (\text{sync, tbl}) r_2. \]
\[ \text{sync}(\_, \_, \text{bijlookup}(\text{bijinvert}(\text{!tbl})) r_2) \]
\[ \text{pubwrap} \triangleq \lambda m. \text{membrane}(\text{pubout m})(\text{pubin m}) \]
\[ \text{pubunwrap} \triangleq \lambda m. \text{membrane}(\text{pubin m})(\text{pubout m}) \]
\[ \text{pubref} \triangleq \lambda m x_1. \]
\[ \text{let r_1} = \text{ref x_1 in} \]
\[ \text{let r_2} = \text{ref (pubwrap m x_1) in} \]
\[ \text{let (sync, tbl)} = m \text{ in} \]
\[ \text{sync}(\_, \_, \text{bijlookup}(\text{!tbl}) r_1); \]
\[ \text{r_1} \]
\[ \text{shadowread} \triangleq \lambda m r. \text{pubunwrap m (! (pubout m r))} \]
\[ \text{shadowwrite} \triangleq \lambda m r x. \text{pubout m r} \leftarrow \text{pubout m x} \]

Fig. 10. Public membrane implementation

We discuss the straightforward functions \( \text{makesync}, \text{bijempty}, \text{bijinsertnew}, \text{bijlookup}, \) and \( \text{bijinvert} \) in Appendix F.
isprivval \( y \ v \triangleq \text{lift (isprivloc } y) \ v \) MakePubSpec
\( \{ T \} \ makepub () \{ m \ y, \ \text{ret} \ m. \ \text{ismembrane} \ y \ m \} \)

PubAllocSpec
\( \{ \text{ismembrane} \ y \ m * \ \text{isprivval} \ y \ v \} \ \text{pubref} \ m \ v \{ \ell, \ \text{ret} \ l. \ \text{isprivloc} \ y \ l * \ \ell \ \rightarrow \ v \} \)

PubWrapSpec
\( \{ \text{ismembrane} \ y \ m * \ \text{isprivval} \ y \ v \} \ \text{pubwrap} \ m \ v \{ x_2, \ \text{lowval} \ x_2 \} \)

PubUnwrapSpec
\( \{ \text{ismembrane} \ y \ m * \ \text{lowval} \ v_2 \} \ \text{pubunwrap} \ m \ v_2 \{ x_1, \ \text{isprivval} \ y \ x_1 \} \)

ShadowReadSpec
\( \{ \text{ismembrane} \ y \ m * \ \text{isprivloc} \ y \ l \} \ \text{shadowread} \ m \ l \{ x. \ \text{isprivval} \ y \ x \} \)

ShadowWriteSpec
\( \{ \text{ismembrane} \ y \ m * \ \text{isprivloc} \ y \ l * \ \text{isprivval} \ y \ v \} \ \text{shadowwrite} \ m \ l \ v \{ \text{ret} (). \ T \} \)

Fig. 11. Public membrane interface

using shadowread and shadowwrite. Appendix E shows an example of how the public membrane is used.

Public membrane specification. The OCPL specification of the public membrane is shown in Fig. 11. The assertion ismembrane \( y \ m \) associates a public membrane \( m \) to a logical name \( y \). The abstract predicate isprivloc \( y \ l \) represents knowledge that location \( l \) is a private location for the membrane named \( y \), i.e., \( l \) was previously allocated using pubref on the public membrane named \( y \). The assertion isprivval \( y \ v \) is defined as lift (isprivloc \( y \) \( v \)) by rule IsPrivval. In particular, the assertion means that any location “extractable” from \( v \) is in the domain of \( y \)’s table. We call such values \( v \) private values.\(^{13}\)

The specification rules are straightforward, so we explain only some salient points here. The triple for pubref (rule PubAllocSpec) says that the application pubref \( m \ v \) returns a location \( l \ \rightarrow \ v \) together with the knowledge that \( l \) is a private location, represented by isprivloc \( y \ l \). It subsequently becomes the responsibility of the public membrane’s client, who owns \( l \ \rightarrow \ v \), to protect any high-integrity state reachable from \( l \) —for example, by keeping \( l \) hidden as we do in Appendix E. The triples PubWrapSpec and PubUnwrapSpec say that the function pubwrap \( m \) converts private values to low values, while pubunwrap \( m \) does the opposite. In particular, the output of pubwrap, being low, can be freely shared with untrusted code. Also, pubunwrap has a non-progressive triple because unwrapping an arbitrary low value \( v_2 \) may apply pubin \( m \ \ell_2 \) with some low-integrity location \( \ell_2 \) unknown to \( m \), and our implementation of pubin gets stuck in such cases.\(^{14}\)

The implementation of the public membrane satisfies this specification. As an example, we briefly describe how we verify the pubwrap specification, PubWrapSpec. First, we prove the following two lemmas showing that for a valid membrane \( m \), pubout \( m \) and pubin \( m \) produce functions that satisfy

\(^{13}\)Reasoning with private values is fairly straightforward using the definition in Fig. 3 and derived rules like LiftApp (§2.2).

\(^{14}\)For simplicity, we do not specify the functionality of pubwrap and pubunwrap beyond “returns a low, resp., private value”. More useful specifications would allow one to prove that, say, after wrapping private location \( \ell_1 \) twice and obtaining values \( v_2 \) and \( v_2’ \), there exists a low location \( \ell_2 \) such that \( v_2 = \ell_2 = v_2’ \).
the predicate \textit{ismon}.

\[
\{\text{ismembrane } \gamma m\} \text{ pubout } m \{f. \text{ ismon progress } f (\text{isprivloc } \gamma) \text{ lowloc}\}
\]

\[
\{\text{ismembrane } \gamma m\} \text{ pubin } m \{f. \text{ ismon noprgrass } f \text{ lowloc } (\text{isprivloc } \gamma)\}
\]

Next, we instantiate \texttt{MembraneSpec} (Fig. 9) with \(\Psi_1 = \text{isprivloc } \gamma\), \(\Psi_2 = \text{lowloc}\) and the two lemmas above to obtain

\[
\text{ismon progress pubwrap } (\text{isprivval } \gamma) \text{ lowval}
\]

The triple \texttt{PubWrapSpec} follows by unfolding the definition of \textit{ismon} (Fig. 9).

\section{RELATED WORK}

This paper focuses on the formal specification and verification of OCPs. While we know of only rather preliminary prior work in this space, a major point of our paper is to observe that the tools needed for effective verification of OCPs are already to a large extent available—the pieces just need to be assembled properly. In particular, our logic, OCPL, gets significant mileage out of existing verification techniques, notably robust safety and concurrent separation logic. In the following, we discuss related work on robust safety, the (limited amount of) existing work on specification of OCPs, and some other relevant work on object capabilities and ownership types.

\textbf{Robust safety.} The concept of robust safety arose in the context of verifying security protocols that interact with adversaries. Early work used typing to prove “correspondence properties” for cryptographic protocols modeled in the spi calculus (Gordon and Jeffrey 2001). In their work on the refinement type-checker F7, Bengtson et al. (2011) generalized robust safety to a richer class of integrity properties for a process calculus, RCF, with higher-order state. We inherit from their work the basic idea of using a notion of low-integrity values and proving robust safety, but the approaches differ greatly in detail. First, we show how to apply this idea to OCPs, a completely different domain. Second, we show how low-integrity values are directly encodable in modern separation logics, using a simple logical relation.

\textbf{Object capabilities.} There has been only very preliminary work on specifying and verifying functional properties of object capabilities and OCPs. In his seminal paper on dynamic sealing, Morris (1973) proposed informal reasoning principles for programmers using dynamic sealing, but did not prove anything formally. Drossopoulou et al. (2015a) proposed predicates modeling trust and risk and used those predicates to specify a capability-based escrow exchange example (Miller et al. 2013). They focused on this one example, whereas we develop general specifications for several OCPs. Further, they focus on syntactic specifications and do not define the semantics of their predicates. A subsequent manuscript (Drossopoulou et al. 2015b) bridges this gap to the semantics, but their Hoare logic seems inadequate for the examples we consider, e.g., it lacks a rule for dynamic allocation.

Perhaps the most closely related work to our own is that of Devriese et al. (2016). As discussed in the introduction, they use a Kripke logical relation and a meta-property called effect parametricity to verify integrity properties for several examples of capability-wrapped user code in a language with higher-order state. There are several points of difference between our work and theirs. First, we work in a concurrent separation logic rather than directly in a low-level logical relation. As a result, in addition to being able to conduct our proofs at a much higher level of abstraction, we can give compositional specifications for higher-order object capability patterns (i.e., libraries), whereas they only verify specific programs that use object capabilities. We also exploit the notion of robust safety in verifying integrity properties of code that uses OCPs, whereas corresponding arguments only appear implicitly in their proofs. On the other hand, they develop semantic variants of the
so-called "reference graph properties" from the literature on object capabilities. These properties are important but they are also orthogonal to verification and, hence, we do not examine them. Last but not least, our proofs are machine-checked in Coq.

While specifying OCPs has received little attention so far, there has been a lot of other work on object capabilities and OCPs. Responding to renewed interest in dynamic sealing, Sumii and Pierce (2004) propose a bisimulation for proving contextual equivalences in a language with a dynamic sealing primitive. Roughly, the examples they consider are pairs of expressions—"modules" implementing the same "interface" using sealing—and the question they study is whether those expressions are indistinguishable, even if their internal representations differ. Bengtson et al. (2011) use dynamic sealing to define ideal implementations of cryptographic operations in RCF. They offer no general specifications for dynamic sealing analogous to the specification in §3, but derive instances of such a specification as needed. Van Cutsem and Miller (2013, §§4.3–4.4) describe a variant of our public membrane, which they use to implement proxy objects supporting so-called language invariants (such as the reasoning enabled, in JavaScript, by the ability to "freeze" object properties). Their focus is on the use of such membranes as an implementation technique, rather than on verification. Spiessens and Van Roy (2005); Spiessens (2007) and Murray (2010) use, respectively, model- and refinement-checking tools to establish certain safety and liveness properties of abstract models of object capability systems, including some OCPs. To take one small example, Murray shows that a specific model of an unsealing operation does not reveal a sealed value unless a capability to that sealed value was passed (possibly indirectly) to the unsealing operation by the context. Although useful, such properties cannot be directly used to verify clients of the OCPs. In contrast, our goal is very different: we write compositional specifications for concrete implementations of OCPs and our specifications can be directly used to verify clients.

Ownership types. There are interesting similarities between the core mechanisms of OCPL and prior work on ownership types (Clarke et al. 2013). In their seminal paper, Clarke et al. (1998) proposed a type system based on what came to be called owners-as-dominators. Roughly, the idea is to impose an ownership relation on objects, separating a public owner object from its hidden representation objects and ensuring that an owner mediates all outside access to its representation. Clarke et al. then proved that well-typed code preserves an invariant classifying data and enforcing isolation according to owners-as-dominators. In a similar vein, OCPL classifies locations as high-integrity (private) vs. low-integrity (public), and enforces that high-integrity locations cannot be accessed from low-integrity locations. However, OCPL is not a static type system, but rather a logic for verifying the enforcement of integrity properties in a dynamically-typed setting. Moreover, although OCPL requires that low locations contain low values, it does not stipulate any particular ownership discipline (like owners-as-dominators) for how the user should achieve this.

Other researchers have proved deeper semantic properties of owners-as-dominators. Banerjee and Naumann (2005a,b) prove relational parametricity theorems, using logical relations as a proof technique for showing that one representation may be safely replaced by another. Patrignani et al. (2011) show that an extension of owners-as-dominators can be used to prove relational secrecy properties for the join calculus, based on a distinction between high- and low-integrity values. In OCPL, we define low-integrity values using a simple (implicitly step-indexed) logical relation, and (like Patrignani et al.) our key meta-theorem concerns the interaction of user code with untyped, untrusted code. However, despite these similarities to prior work, our concrete goals are very different: rather than reasoning about relational properties, we focus on verifying functional specifications of OCPs and the robust safety of capability-wrapped code.
CONCLUSION AND FUTURE WORK

Object capability patterns (OCPs) enable programmers to enforce invariants on the private state of their objects, even when those objects are passed to untrusted code. In principle, this should make it easier to write secure and correct programs, but in practice, programmers may use OCPs incorrectly, resulting in subtle security flaws. In this paper, we develop OCPL, the first logic for compositionally specifying and verifying OCPs. We deploy it in reasoning about both implementations and representative clients of several well-known OCPs, in the context of a simple but expressive programming language with higher-order state.

We believe that using robust safety and separation logic to reason about OCPs scales to much richer settings. The Firefox web browser, for example, uses an automatic and significantly more sophisticated membrane pattern to enforce the so-called same-origin policy (Mozilla 2016; Barth 2011). In ongoing work, we are exploring an extension of OCPL to reason about this system. Furthermore, while the Iris proof mode enables relatively high-level Coq proofs, such manual proofs are nonetheless often tedious and routine. Additional research is needed to scale such proofs up to realistic languages and improve automation.

ACKNOWLEDGMENTS

This research was supported in part by a Microsoft Research PhD Scholarship, and 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). We wish to thank Andrea Fischer and the OOPSLA reviewers for extremely helpful and detailed feedback.

REFERENCES


OCPL 2017. Long version of this paper (with appendices) and Coq development. (July 2017). Available at the Iris project website at http://plv.mpi-sws.org/iris/.


A SEALING WITH LOW FUNCTIONS

In this appendix, we point out that, in the dynamic sealing OCP, representing proxy values as low-integrity functions renders unsealing non-deterministic, forcing somewhat surprising changes to otherwise “obviously safe” code.

Our implementation of sealing differs from Morris’ in one important respect. Ignoring synchronization, Morris sealed a value \( v \) by returning a function \( \lambda k. \, tbl \leftarrow \text{mapinsertnew}(!tbl \, k \, v) \) for extending the table.\(^{15}\) To unseal a value \( f \), Morris allocated a fresh location \( k \), applied \( f \, k \), and looked for \( k \) in the table. This simple change—returning a function rather than a location—renders unsealing non-deterministic, forcing somewhat surprising changes that, when unsealed, oscillates between \( v_1 \) and \( v_2 \). An easy fix is to weaken the triple further, giving \( f \) that, when unsealed, oscillates between \( v_1 \) and \( v_2 \). An easy fix is to weaken the triple further, giving up on deterministic unsealing.

We verified Morris’ implementation of sealing against a weaker specification, obtained by replacing rule \text{UnsealAnySpec} in Fig. 5 with

\[
\text{UnsealLowSpec} \defeq \{ \text{isunseal} \, y \, u \, \phi \} \, u \, v' \{ x. \, \text{issealed} \, y \, x \, v' \, \phi \},
\]

This switch to non-deterministic unsealing of low values comes at some cost. We must weaken the non-progressive triples in our intervals specification; for example, replacing \text{MinAnySpec} by

\[
\text{MinLowSpec} \defeq \{ \text{ismin} \, y \, \text{imin} \, \ast \, \text{lowval} \, v \} \, \text{imin} \, v \{ n, \text{ret} \, n. \, \top \},
\]

With these weaker rules, we can no longer verify the function \( \text{check} = \lambda j. \, \text{assert} \, (\text{imin} \, j \leq \text{imax} \, j) \) in our intervals client. Applying \( \text{check} \) to a monster like the oscillating function \( f \), we know by assumption that we can apply \( f \, k \) (if we can mark \( k \) low—we can). Second, the triple returns a value \( v \) such that issealed \( y \, v \, v' \, \phi \) which means that subsequent applications \( u \, v' \) must also return \( v \). This is absurd! An adversary with values \( f_1, f_2 \) obtained by sealing values \( v_1, v_2 \) can, for example, construct

\[
f \triangleq \text{let } r = \text{ref} \, \text{false} \, \text{in} \, \lambda k. \, \text{let } b = !r \, \text{in} \, (r \leftarrow \text{not} \, b; (\text{if } b \, \text{then } f_1 \, \text{else } f_2 \, k))
\]

that, when unsealed, oscillates between \( v_1 \) and \( v_2 \). An easy fix is to weaken the triple further, giving up on deterministic unsealing.

\(^{15}\)We are not so much interested in Morris’ implementation but in the fact that it returns a function. Other dynamic sealing implementations return functions and otherwise have attractive properties (e.g., dispensing with the table). Ignoring synchronization, the table-free implementation of sealing (Miller 2008; Taly et al. 2011)

\[
\text{let } r = \text{ref} \, \text{None} \, \text{in} \, \text{let } \text{seal} = \lambda x_\rightarrow. \, r \leftarrow \text{Some} \, x \, \text{in} \, \text{let } \text{unseal} = \lambda f. \, (r \leftarrow \text{None}; \, \text{valOf} \, !r) \, \text{in} \, (\text{seal}, \, \text{unseal})
\]

shares this drawback with Morris’ implementation.
Asymmetric signatures.

\[
\text{MakeSignSpec} \quad \phi \text{ persistent} \quad (\forall \nu) \phi \nu \vdash \text{lowval} \nu
\]

\[
\{ \top \}
\]

\text{makeseal} ()

\[
\{ v_1 v_2 \gamma, \text{ret} (v_1, v_2). \text{issign} \gamma v_1 \phi \ast \text{isverify} \gamma v_2 \phi \}
\]

\text{SignSpec}

\[
\{ \text{issign} \gamma \text{sign} \phi \ast \phi \nu \} \text{sign} \nu \{ x'. \text{issigned} \gamma \nu x' \phi \}
\]

\text{VerifySpec}

\[
\{ \text{isverify} \gamma \text{verify} \phi \ast \text{issigned} \gamma v \nu' \phi \} \text{verify} \nu' \{ \text{ret} \nu, \top \}
\]

\text{VerifyAnySpec}

\[
\{ \text{isverify} \gamma \text{verify} \phi \} \text{verify} \nu' \{ x. \text{issigned} \gamma x x' \phi \}, \quad \text{isverify} \gamma \text{verify} \phi \vdash \text{lowval} \text{verify}
\]

\text{SignedLow}

\[
\text{issigned} \gamma v \nu' \phi \vdash \text{lowval} \nu'
\]

\text{SignedInv}

\[
\text{issigned} \gamma v \nu' \phi \vdash \phi \nu
\]

\text{SignedAgree}

\[
\text{issigned} \gamma v_1 v' \phi \ast \text{issigned} \gamma v_2 v' \phi \vdash v_1 = v_2
\]

Asymmetric encryption.

\text{MakeEncryptSpec}

\[
\{ \top \}
\]

\text{makeseal} ()

\[
\{ v_1 v_2 \gamma, \text{ret} (v_1, v_2). \text{isencrypt} \gamma v_1 \ast \text{isdecrypt} \gamma v_2 \}
\]

\text{EncryptSpec}

\[
\{ \text{isencrypt} \gamma \text{enc} \} \text{enc} \nu \{ x'. \text{isctext} \gamma x x' \}
\]

\text{DecryptSpec}

\[
\{ \text{isdecrypt} \gamma \text{dec} \ast \text{isctext} \gamma \nu \nu' \} \text{dec} \nu' \{ \text{ret} \nu, \top \}
\]

\text{DecryptAnySpec}

\[
\{ \text{isdecrypt} \gamma \text{dec} \} \text{dec} \nu' \{ x. \text{isctext} \gamma x x' \}', \quad \text{isencrypt} \gamma \text{enc} \vdash \text{lowval} \text{enc}
\]

\text{CtextLow}

\[
\text{isctext} \gamma v \nu' \vdash \text{lowval} \nu'
\]

\text{CtextAgree}

\[
\text{isctext} \gamma v_1 v' \ast \text{isctext} \gamma v_2 v' \vdash v_1 = v_2
\]

\text{Fig. 12. Public-key interfaces (derived from Fig. 5)}

one:

\text{SnapSpec}

\[
\{ \text{issnap} \gamma \text{snap} \ast \text{isinterval} \gamma n_1 n_2 \nu \} \text{snap} \nu \{ i. \text{isinterval} \gamma n_1 n_2 i \}
\]

\text{SnapLowSpec}

\[
\{ \text{issnap} \gamma \text{snap} \ast \text{lowval} \nu \} \text{snap} \nu \{ i n_1 n_2, \text{ret} i. \text{isinterval} \gamma n_1 n_2 i \},
\]

\text{SnapLow}

\[
\text{issnap} \gamma \text{snap} \vdash \text{lowval} \text{snap}
\]

With these changes, we can prove the assertion expression in our modified intervals client succeeds and the client is robustly safe.
B PUBLIC-KEY INTERFACES FOR SEALING

In this appendix, we show that dynamic sealing can be used to implement ideal cryptographic signing/verification as well as encryption/decryption primitives, and that the specification of dynamic sealing (Fig. 5) can be specialized to derive very general specifications for these primitives. Such ideal primitives are sometimes used in the verification of cryptographic protocols (Bengtsson et al. 2011). While the observation that dynamic sealing can implement these cryptographic primitives is not new—Morris (1973) notes this even though he did not use the terms signing and encryption back then—the observation that specifications for the cryptographic primitives can be derived from those for dynamic sealing is new.

Signature/verification. A signature/verification primitive provides a function sign that creates a low proxy for its argument. This proxy is usually called a signature. The proxy (signature) can be verified using a dual function verify, which when applied to the value returned by sign \( v \), returns \( v \) and if the argument was not the output of sign, gets stuck. The important point is how sign and verify are used in practice. In typical use, sign is held private by a piece of code that has been verified to apply sign only to values that satisfy a representation invariant \( \phi \). The verify function is publicly available to everyone and anyone can use it to verify signatures created by the code that holds sign. In this mode of use, signing/verification transfers knowledge of the representation invariant from the signer to the verifier: since the signer only signs values with representation invariant \( \phi \), and verify only returns previously signed values, the verifier always knows that any value returned by verify must satisfy \( \phi \).

It should be clear that sign and verify are quite similar to seal and unseal returned by makeseal(). The top half of Fig. 12 shows that, in fact, seal and unseal are an implementation of sign and verify. The specification shown in this figure is a specific instance of the general sealing specification from Figure 5, derived by defining the abstract predicates issign, isverify, and issigned to be isseal, isunseal, and issealed, respectively; using UnsealLow to prove VerifyLow; and dropping SealLow. The derived specification says that makeseal() returns a pair of functions that behave like sign and verify if the representation invariant \( \phi \) implies lowval. This condition is necessary since verify returns previously signed values to untrusted code, so signed values should always be low—in cryptographic terms, signatures provide no secrecy. Additionally, the specification says that the value returned by verify always satisfies \( \phi \) (rule VerifyAnySpec), that verify itself is low (rule VerifyLow) so verify can be shared safely with untrusted code, and that the output of sign is low (rule SignedLow) so signatures can also be shared safely with untrusted code. On the other hand, sign itself is not low, so it should not be shared with untrusted code.

Encryption/decryption. An encryption/decryption primitive provides a function enc that creates a low proxy for its argument. A dual function dec converts the proxy back to the original argument. Typically, encryption/decryption is used by holding dec private in some piece of code, and making enc public. This allows anyone to encrypt a secret using enc and share it freely, with the guarantee that only the code holding dec can ever access the secret.

Again, enc and dec look similar to seal and unseal returned by makeseal() and the bottom half of Fig. 12 formalizes this intuition by specializing the specification of Fig. 5 to match the intuitive description of encryption/decryption. The abstract predicates isencrypt, isdecrypt, and isctxtare defined to be isseal, isunseal, and issealed, respectively. Importantly, the specification says that makeseal() returns functions that behave like enc and dec (rule MakeEncryptSpec), that decrypting a previously encrypted value \( v \) returns \( v \) (rules EncryptSpec and DecryptSpec), and that the encryption function and ciphertexts are low (rules EncryptLow and CtextLow) so they can be shared with
The following rules constitute the full specification of the intervals library in §3.

**Progressive triples.**

\[
\begin{align*}
\text{MAKEINT} & \quad \text{MAKEINT'} \\
\{ \text{ismakeint } y \ mk \} & \quad \{ \text{ismakeint' } y \ n_1 \ f \} \\
\text{SUM} & \quad \text{MIN} \\
\{ \text{issum } y \ isum \} & \quad \{ \text{imin } y \ imin \} \\
\{ \text{issum' } y \ n_1 \ n_2 \ f \} & \quad \{ \text{imin' } y \ n_1 \ n_2 \} \\
\text{INTVL} & \quad \text{INTVLINV} \\
\{ \text{ismakeint } y \ v_1 \} & \quad \{ \text{ismakeint' } y \ n_1 \ f \} \\
\end{align*}
\]

**Non-progressive triples.**

\[
\begin{align*}
\text{MINANY} & \quad \text{MAXANY} \\
\{ \text{imin } y \ imin \} & \quad \{ \text{imax } y \ imax \} \\
\{ \text{imin' } y \ n_1 \ n_2 \} & \quad \{ \text{imax' } y \ n_1 \ n_2 \ f \} \\
\text{SUMANY} & \quad \text{SUMANY'} \\
\{ \text{issum } y \ isum \} & \quad \{ \text{issum' } y \ n_1 \ n_2 \ f \} \\
\text{SUM'ANY} & \quad \text{SUM'ANY'} \\
\{ \text{issum' } y \ n_1 \ n_2 \ f \} & \quad \{ \text{issum' } y \ n_1 \ n_2 \ f \} \\
\end{align*}
\]

**Low values.**

\[
\begin{align*}
\text{INTVLLOW} & \quad \text{MAKEINT'LOW} & \quad \text{MAKEINTYLOW} \\
isinterval y \ n_1 \ n_2 \ i \vdash \text{lowval } i & \quad \text{ismakeint' } y \ n_1 \ f \vdash \text{lowval } f & \quad \text{ismakeint } y \ mk \vdash \text{lowval } mk
\end{align*}
\]
D BLOCKING CARETAKERS

In this appendix, we briefly present a simpler implementation of API caretakers.\footnote{We discuss the straightforward functions \textit{makelock'}, \textit{release}, \textit{acquire}, and \textit{syncwith} in Appendix F.}

\begin{align*}
\text{makecaretaker} & \triangleq \text{makelock'} & \text{enable} & \triangleq \text{release} \\
\text{wrap} & \triangleq \lambda ct \, f \, x. \text{syncwith} \, ct \, (\lambda_{-} \, f \, x) & \text{disable} & \triangleq \text{acquire}
\end{align*}

The idea is that, if we are happy to let wrappers block until a caretaker is enabled, we can dispense with the \textit{enabled} flag used in §4. In this implementation, a caretaker is a lock. The function \textit{makelock'} allocates a fresh lock, initially locked. The function \textit{syncwith} is analogous to \textit{sync}, except that it takes the lock to use as an initial argument. To disable and enable wrappers, we simply acquire and release the lock they need. This implementation also satisfies the API caretaker specification (Fig. 7).

E PUBLIC MEMBRANE CLIENT

In this appendix, we further illustrate the public membrane by describing a simple client program that uses it. First, we define the following auxiliary function, \textit{getint}. This function presupposes that we have some invariant on a private integer location, \(r\), and that we want to periodically update \(r\) by "merging" its contents with those of its shadow location, using some merge function \(f\) that preserves the invariant.

\[
\text{getint} \triangleq \lambda m \, f \, r. \, \text{let} \, n_1 = \text{!}r \, \text{in} \, \text{let} \, x_2 = \text{shadowread} \, m \, r \, \text{in} \\
\text{if} \, \text{isint} \, x_2 \, \text{then} \\
\quad \text{let} \, n_3 = f \, n_1 \, x_2 \, \text{in} \\
\quad \text{let} \, _{-} = \text{if} \, n_1 \neq n_3 \, \text{then} \, r \leftarrow n_3 \, \text{else} () \, \text{in} \\
\quad \text{let} \, _{-} = \text{if} \, x_2 \neq n_3 \, \text{then} \, \text{shadowwrite} \, m \, r \, n_3 \, \text{else} () \, \text{in} \\
\quad n_3 \\
\text{else} \, (\text{shadowwrite} \, m \, r \, n_1; n_1)
\]

If the private location \(r\) and its shadow contain integers \(n_1\) and \(n_2\), the code applies \(f \, n_1 \, n_2\) to obtain the "merged" integer \(n_3\), updates \(r\) and its shadow, and returns \(n_3\); otherwise, the code overwrites the shadow’s contents with (and returns) \(n_1\).

Our illustrative client program is a private up/down counter with increment and decrement functions, and private lower and upper limits. The invariant we have in mind is that the counter always remains within these limits. However, the counter also exposes shadow locations for the lower and upper limits to its untrusted context. The context can freely update these shadow lower and upper limits. However, these changes are propagated to the private locations (using \textit{getint})
Theorem RobustSafety applied to this triple. This follows because

\( \text{lo} \) \text{client} \ The \ client \ CAS \ example, the evident implementation of spin locks using verification details to our Coq development. In this appendix, we specify the implementations of locks, finite maps, and finite bijections used in F LIBRARY FUNCTIONS

\( \text{client} \triangleq \text{let } m = \text{makepub}() \text{ in } \)
\( \text{let } lo = \text{pubref} m 0 \text{ in } \text{let } c = \text{ref} 0 \text{ in } \text{let } hi = \text{pubref} m 0 \text{ in } \)
\( \text{let } \text{sync} = \text{makesync()} \text{ in } \)
\( \text{let } \text{use} = \lambda_. \text{sync}(\lambda_. \text{assert} \,!\! \text{lo} \leq \!\! \text{c}; \text{assert} \!\! \text{c} \leq \!\! \text{hi}) \)
\( \text{let } \text{getlimits} = \lambda_. \text{let } n = \text{!} \text{c} \text{ in } \)
\( \text{let } a = \text{getint}(\lambda n_1 n_2. \text{if } n_2 \leq n \text{ then } n_2 \text{ else } n_1) \text{ lo } \text{ in } \)
\( \text{let } b = \text{getint}(\lambda n_1 n_2. \text{if } n \leq n_2 \text{ then } n_2 \text{ else } n_1) \text{ hi } \text{ in } \)
\( (a, b) \)
\( \text{let } \text{decr} = \lambda_. \text{sync}(\lambda_. \text{let } n = \text{!} \text{c} - 1 \text{ in } \)
\( \text{let } b = \text{fst}(\text{getlimits}()) \leq n \text{ in } \)
\( \text{if } b \text{ then } c \leftarrow n \text{ else } (); b) \)
\( \text{let } \text{incr} = \lambda_. \text{sync}(\lambda_. \text{let } n = \text{!} \text{c} + 1 \text{ in } \)
\( \text{let } b = n \leq \text{snd}(\text{getlimits}()) \text{ in } \)
\( \text{if } b \text{ then } c \leftarrow n \text{ else } (); b) \)
\( \text{let } \text{lo}' = \text{pubwrap} m \text{ lo } \text{ in } \text{let } \text{hi}' = \text{pubwrap} m \text{ hi } \text{ in } \)

\( (\text{use, lo}', \text{hi}', \text{incr}, \text{decr}) \)

The \text{client}’s private state comprises a public membrane \text{m}, a private counter \text{c} and private limits \text{lo} and \text{hi}, subject to the representation invariant \!\! \text{lo} \leq \!\! \text{c} \leq \!\! \text{hi} \,(i.e., \text{the counter respects its limits}). The \text{client} returns (i) a simple function \text{use} that, when applied, simply asserts the representation invariant (and that could be extended to offer some service to the counter’s clients), (ii) the shadow locations \text{lo}' and \text{hi}' that substitute \text{lo} and \text{hi} for the surrounding untrusted code (the untrusted code may freely change these at any time) and (iii) increment and decrement functions, \text{incr} and \text{decr}, that either change the counter \text{c} and return \text{true} or (when changing the counter would violate its current limits) return \text{false}. Prior to changing the counter, \text{incr} and \text{decr} copy the shadow locations \text{lo}' and \text{hi}' into \text{lo} and \text{hi} using the function \text{getint}, if doing so will not break the representation invariant.

Using the rules of OCPL, we prove that \text{client} is robustly safe, meaning that no context can exploit the returned values \text{use}, \text{incr}, \text{decr}, \text{lo}' and \text{hi}' in a way that violates the assertions about the representation invariant in \text{use}. To do this, we first prove that \text{client} satisfies the following triple: \{ \text{\top} \} \text{client} \{ \text{x. lowval x} \}. One key step in this proof is showing that \text{lo}' and \text{hi}' are both low. This follows because \text{lo}' and \text{hi}' are obtained from \text{pubwrap} and the postcondition of \text{pubwrap} (rule PubWrapSpec, Fig. 11) says that the value it returns is low. Robust safety follows immediately from Theorem RobustSafety applied to this triple.

\section*{F \ LIBRARY FUNCTIONS}
In this appendix, we specify the implementations of locks, finite maps, and finite bijections used in our examples. As nothing here is original or terribly interesting, we relegate many implementation and verification details to our Coq development.

\text{Locks and synchronization}. We specify locks and synchronization in Fig. 13. Rules LockedExclusive-AcquireSpec constitute a CAP-style lock specification (Dinsdale-Young et al. 2010) satisfied by, for example, the evident implementation of spin locks using CAS. The assertion islock γ lk R represents
Initially locked locks.

\textbf{LockedExcluisive}
\begin{align}
\text{locked } \gamma \rightarrow \text{locked } \gamma \not\vdash \bot
\end{align}

\textbf{MakeLock'}Spec
\begin{align}
\{ \top \} \text{ makelock'}() \{ lk \gamma, \text{ret} lk. \text{islock} \ y \ lk \ R \not\vdash \text{locked } \gamma \}
\end{align}

\textbf{ReleaseSpec}
\begin{align}
\{ \text{islock} \ y \ lk \ R \not\vdash \text{locked } \gamma \ R \} \text{ release} lk \ \{ \text{ret}(). \top \}
\end{align}

\textbf{AcquireSpec}
\begin{align}
\{ \text{islock} \ y \ lk \ R \} \text{ acquire} lk \ \{ \text{ret}(). \text{locked } \gamma \ R \}
\end{align}

Initially unlocked locks.

\textbf{MakeLockSpec}
\begin{align}
\{ R \} \text{ makelock}() \{ lk \gamma, \text{ret} lk. \text{islock} \ y \ lk \ R \}
\end{align}

\textbf{Synchronization.}
\begin{align}
\text{IsSync}
\text{issync} \text{ sync} \ R \triangleq \square \forall Q. \{ R \} \ e \ \{ v. \ R \not\vdash Q \} \rightarrow \text{SyncWithSpec}
\begin{align}
\{ \top \} \text{ sync}(\lambda_\_ e) \ \{ v. \ Q \} \quad \{ \text{islock} \ y \ lk \ R \} \text{ syncwith} lk \ \{ v. \text{issync} v R \}
\end{align}

\textbf{MakeSyncSpec}
\begin{align}
\{ R \} \text{ makesync}() \ \{ v. \text{issync} v R \}
\end{align}

Fig. 13. Lock and synchronization interface

knowledge that the value \( lk \) is a lock with abstract name \( \gamma \) protecting resource \( R \). The assertion \( \text{locked } \gamma \) represents ownership of the lock named \( \gamma \). Thus, for any resource \( R \),

- \textbullet \ \text{makelock}'() returns a new, locked lock \( lk \) protecting \( R \) (\text{MakeLock'}Spec);
- \textbullet \ \text{release} \ lk \ gives up lock \( lk \) and resource \( R \), returning unit (\text{ReleaseSpec}); and
- \textbullet \ \text{acquire} \ lk \ grabs lock \( lk \) and the resource \( R \) it protects, returning unit (\text{AcquireSpec}).

This specification is atypical in that \text{makelock}' constructs a locked lock, a feature needed only to argue that our caretaker specification is just a specialized lock spec (see the implementation of “blocking caretakers” in Appendix D). \text{MakeLockSpec} embodies a typical CAP-style lock specification: it specifies the evident wrapper around \text{makelock}' and \text{release} that consumes resource \( R \) in order to construct a new, unlocked lock \( lk \) protecting resource \( R \).

Rules \text{IsSync}–\text{MakeSyncSpec} specify the following simple library, readily verified against any lock implementation.

\begin{align}
\text{syncwith} \triangleq \lambda lk \ f. \ \text{acquire} \ lk; \ \text{let} \ r = f() \ \text{in} \ \text{release} \ lk; \ r \\
\text{makesync} \triangleq \lambda_\_. \ \text{let} \ lk = \text{makelock}() \ \text{in} \ \text{syncwith} \ lk
\end{align}

Given a lock \( lk \) and a thunk \( f \), the function \text{syncwith} applies \( f \) while holding the lock, returning whatever \( f() \) returns. The function \text{makesync} returns \text{syncwith} \( lk \) for a fresh lock \( lk \). In specifying \text{makesync}, we follow Turon et al. (2013). Definition \text{IsSync} is the key. It says that to prove \text{sync}(\lambda_\_ e) returns a value satisfying \( Q \), it suffices to show that \( e \), when given access to the resource \( R \) protected by \text{sync}'s lock, (i) returns a value satisfying \( Q \) and (ii) preserves \( R \).

Finite maps. We specify functional finite maps in Fig. 14. The spec is parameterized by an injection \( \iota: X \rightarrow \text{Val} \) from some type \( X \) (our examples use locations) to values. The assertion

\text{ismap} \ f
### Finite maps

\begin{align*}
\text{MapEmptySpec} & \\
ismap \text{mapempty} \emptyset
\end{align*}

\begin{align*}
\text{MapInsertNewSpec} & \\
\{ \text{ismap map} \} \text{mapinsertnew map}(\mathit{i}x) \mathit{v} \{ \text{map’}. \text{ismap map’} \mathit{f}[\mathit{x} \mapsto \mathit{v}] \}
\end{align*}

\begin{align*}
\text{MapLookupPartialSpec} & \\
\{ \text{ismap map} \} \text{maplookup map}(\mathit{i}x) \{ \mathit{y}. \mathit{f} \mathit{x} = \mathit{y} \},
\end{align*}

*Fig. 14. Finite map interface*

\begin{align*}
\text{BijEmptySpec} & \\
isbij \text{bijempty} \emptyset \emptyset
\end{align*}

\begin{align*}
\text{BijInvertSpec} & \\
\{ \text{isbij bij} \} \text{bijinvert bij} \{ \text{bij’}. \text{isbij bij’} \mathit{f’} \mathit{f} \}
\end{align*}

\begin{align*}
\text{BijInsertNewSpec} & \\
\{ \text{isbij bij} \mathit{f’} \mathit{f} \} \text{bijinsertnew bij}(\mathit{i}x) \{ \mathit{f’} \mathit{x} \mapsto \mathit{i}x \}
\end{align*}

\begin{align*}
\text{BijLookupPartialSpec} & \\
\{ \text{isbij bij} \mathit{f’} \mathit{f} \} \text{bijlookup bij}(\mathit{i}x) \{ \mathit{x’}, \text{ret} \mathit{i}x’. \mathit{f} \mathit{x} = \mathit{i}x’ \mapsto \mathit{f’} \mathit{x’} = \mathit{i}x \}
\end{align*}

*Fig. 15. Finite bijection interface*

denotes knowledge that value map represents the finite partial function \( f : X \to \text{Val} \). MapEmptySpec says that value mapempty represents the empty map. MapInsertNewSpec specifies a special case of insertion. MapLookupPartialSpec specifies a partial lookup operation. The evident implementation of finite maps using association lists satisfies this spec when \( X \) enjoys decidable equality.

**Finite bijections.** We specify functional finite partial bijections in *Fig. 15*. As with finite maps, the spec is parameterized by an injection \( \iota : X \to \text{Val} \) from some type \( X \) to values. The assertion

\begin{align*}
isbij \text{bij} \mathit{f} \mathit{f’}
\end{align*}

denotes knowledge that value bij represents the finite partial bijection \( f, f’ : X \to \text{Val} \) in the sense that for every \( x \in X \) and value \( v’ \), if \( f \mathit{x} = v’ \), then there exists \( x’ \in X \) such that \( v’ = \iota x’ \) and \( f’ x’ = i x \). Representing a bijection as a pair of finite maps (as specified in *Fig. 14*) leads to a natural implementation of this specification.