An Iris Instance for Verifying CompCert C Programs

WILLIAM MANSKY, University of Illinois Chicago, USA
KE DU, University of Illinois Chicago, USA

Iris is a generic separation logic framework that has been instantiated to reason about a wide range of programming languages and language features. Most Iris instances are defined on simple core calculi, but by connecting Iris to new or existing formal semantics for practical languages, we can also use it to reason about real programs. In this paper we develop an Iris instance based on CompCert, the verified C compiler, allowing us to prove correctness of C programs under the same semantics we use to compile and run them. We take inspiration from the Verified Software Toolchain (VST), a prior separation logic for CompCert C, and reimplement the program logic of VST in Iris. Unlike most Iris instances, this involves both a new model of resources for CompCert memories, and a new definition of weakest preconditions/Hoare triples, as the Iris defaults for both of these cannot be applied to CompCert as is. Ultimately, we obtain a complete program logic for CompCert C within Iris, and we reconstruct enough of VST's top-level automation to prove correctness of simple C programs.

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

Additional Key Words and Phrases: program verification, concurrent separation logic, interactive theorem proving, Verified Software Toolchain, Iris

ACM Reference Format:
William Mansky and Ke Du. 2024. An Iris Instance for Verifying CompCert C Programs. *Proc. ACM Program. Lang.* 8, POPL, Article 6 (January 2024), 27 pages. https://doi.org/10.1145/3632848

1 INTRODUCTION

Iris [Jung et al. 2015], a language-independent framework for concurrent separation logic (CSL) proofs in Coq [Coq Development Team 2023], has been the platform for a wide range of recent research projects: on distributed systems [Krogh-Jespersen et al. 2020], weak memory [Dang et al. 2019; Kaiser et al. 2017], crash safety [Chajed et al. 2019], and many more. Most instantiations of Iris use simple core calculi capturing the key features of interest (shared memory, message passing, weak-memory concurrency, etc.), but several model code in (fragments of) real languages, including Go [Chajed et al. 2019], Rust [Jung et al. 2017], C [Sammler et al. 2021], and WebAssembly [Rao et al. 2023]. Most of these instances use ad-hoc language semantics developed for Iris, and in many cases their code cannot even be run—they are relational models of the languages, not executable semantics or interpreters. A notable exception is Iris-Wasm [Rao et al. 2023], which is built on an existing formal semantics derived from the WebAssembly reference interpreter [Watt et al. 2021]. In this paper, we instantiate Iris with the C semantics of CompCert [Leroy 2009], a verified compiler for a large subset of C. This allows us to use Iris to verify real-world C programs that can then be compiled with CompCert and executed, using the same semantics for the separation logic and the verified compilation. This is probably the largest instantiation of Iris with a pre-existing semantics, and certainly the first attached to a verified compiler.

Authors' addresses: William Mansky, mansky1@uic.edu, University of Illinois Chicago, USA; Ke Du, kdu9@uic.edu, University of Illinois Chicago, USA.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/1-ART6
https://doi.org/10.1145/3632848

There is already a Coq-based separation logic verifier for CompCert C: the Verified Software Toolchain (VST) [Appel et al. 2014]. However, compared to Iris, the design of VST is quite monolithic: almost all of its development is specific to C, and Hoare rules are proved directly on the heap model, making it quite hard to maintain and extend. In contrast, the flexible design of Iris has allowed various extensions to both the model (transfinite step-indexing [Spies et al. 2021], later credits [Spies et al. 2022], etc.) and the verification interface (Diaframe [Mulder et al. 2022], Lithium [Sammler et al. 2021], etc.). In this work, we obtain the best of both worlds by rebuilding the program logic of VST on top of Iris, exporting exactly the same Hoare rules and adequacy theorems as VST from our Iris-based logic. Our logic can prove correctness of any program that can be verified with VST, and gives the same guarantee of correctness on executing programs; at the same time, it makes free use of the logical innovations of Iris, and can more easily be connected to other pieces of the extensive Iris ecosystem.

Instantiating Iris with CompCert C presents several theoretical challenges. First, we need a **model** that can substantiate our logic. The model of Iris assertions is quite simple and generic: a map from indices to arbitrary *resource algebras* (also called cameras), describing all the state that assertions in the logic will reason on. For program verification, Iris developments usually include in this map a *heap* algebra relating the physical memory of the program directly to points-to assertions in the logic. In a logic for CompCert C, there is reason to make this relation more complex than usual: CompCert includes *permissions* on each memory location, which have a clear but nontrivial relationship to ownership shares in separation logic, and also treats special values like function pointers differently from standard values.

Second, we need to define the notion of **safety** that will be used to construct our Hoare triples. This involves defining in the logic what it means for a program to satisfy a pre- and postcondition, and is usually stated in terms of the small-step operational semantics of the programming language. In CompCert and VST, the step relation includes a notion of *external function calls*, used to implement separate compilation, concurrency, and other features: these external calls must be reflected in the semantics that we reason over, and taken into account when proving **adequacy** of the logic.

Finally, we must prove **soundness of our program logic rules** against our definition of safety. The outline of these proofs follows VST, but the proofs themselves are significantly different: while VST unfolds the definition of the program logic and reasons directly on representations of logical memories, the Iris approach allows us to prove Hoare rules entirely within the logic itself. Compared to standard Iris developments, VST's proof rules include some unusual complexities: for instance, VST uses a type system in the program logic to ensure that expressions evaluate correctly. C programs also start with some nontrivial state initialized before the beginning of the `main` function (e.g., global variables), which we reflect in a standard precondition for `main` to allow users to reason about this state in their proofs.

In summary, our contributions are:

- We define a resource algebra for heap resources suitable for CompCert's permission model, including a new generic algebra for possibly-unreadable ownership of resources.
- We adapt Iris's definition of weakest preconditions and Hoare triples to our resource model, and to CompCert's C semantics.
- We re-prove the rules of Verifiable C verbatim on top of our Iris-based logic, giving us a program logic in Iris for CompCert C with the same correctness guarantees as VST.
- We adapt enough of VST's automation to prove correctness of simple C programs using our new logic, and demonstrate that the proof scripts are almost unchanged from VST's, except where we may use Iris tactics to make proofs easier.

All definitions and theorems in the paper are formalized in Coq.

The paper is structured as follows. In section 3, we present the model of our logic, and define the basic points-to predicate. In section 4, we define safety via a weakest precondition assertion, and prove the adequacy theorem that relates the logic to CompCert's semantics. In section 5, we re-prove the Hoare rules of VST on top of our logic, giving us a working program logic for C. In section 6, we apply our logic to some of VST's example programs. In section 7.1, we discuss the structure of our Coq development and compare it to the corresponding components of VST. In sections 8 and 9, we review related work and discuss potential applications and extensions.

A note on concurrency. Separation logics in Iris are naturally concurrent, but CompCert's semantics are strictly sequential. Thus, while the program logic we present in this work could easily be adapted to concurrency, our adequacy result only applies to sequential programs, and our logic should be considered a sequential logic. We discuss in section 9.1 some possible pathways to a concurrent separation logic for (a concurrent extension of) CompCert C.

2 BACKGROUND

2.1 Iris

Iris [Jung et al. 2018] is a language-independent separation logic framework implemented in the Coq theorem prover [Coq Development Team 2023]. Its key innovation is a very generic formulation of *ghost state*, abstract logical state that is used in proofs to model features of the program under consideration that are not explicit in the semantics, such as protocols for access to shared resources. Iris defines a form of step-indexed resource algebra, called a *camera*, that behaves "enough like a heap" to be targeted by separation logic assertions, and then defines a generic separation logic on an arbitrary collection of cameras. Crucially, because step-indexing is built into the definition of cameras, they can be used as *higher-order* ghost state: they can contain program logic assertions, Hoare triples, state machines where each state is associated with an assertion, and a wide range of other useful constructions.

$$(a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad a \cdot b = b \cdot a \qquad |a| \cdot x = x \qquad ||a|| = |a|$$

$$a \leq b \Rightarrow |a| \leq |b| \qquad \checkmark(a \cdot b) \Rightarrow \checkmark a$$

$$\text{where } a \leq b \triangleq \exists c.\ b = a \cdot c$$

Fig. 1. Iris cameras

A camera consists of a carrier set S, a *validity predicate* $\checkmark$, a *core* operation $|\cdot|$, and an operator $\cdot$ that together satisfy the rules shown in Figure 1. The key feature is the associative, commutative operator $\cdot$, which is used to implement separating conjunction of resources. The core $|x|$ of a resource represents *persistent* information that can be arbitrarily duplicated once learned: for instance, in an algebra of transition systems, $|q|$ represents the knowledge that the system is in a state reachable from state q. Finally, in separation logic proofs, we are only allowed to own *valid* elements of the camera: this allows us to rule out the coexistence of particular elements, e.g., to declare that two threads cannot both have full ownership of the same memory location.

Cameras are injected into the separation logic via the *ghost state ownership* assertion $\boxed{a}^{\gamma}$, where a is an element of a camera and γ is a *ghost name* where the element is held (analogous to a memory location in a physical heap). Some of the rules of ghost state ownership are shown in Figure 2. We can allocate new ghost state containing any valid element a of a camera at any point in a program. We can also change the value of ghost state using a *frame-preserving update*, written as $\rightsquigarrow$: we can change a to b as long as any frame c consistent with a is also consistent with b, i.e., the change

$$\left[\overline{a}\right]^{\gamma} * \left[\overline{b}\right]^{\gamma} \dashv\vdash \left[\overline{a \cdot b}\right]^{\gamma} \qquad \left[\overline{a}\right]^{\gamma} \vdash \checkmark a$$

$$\text{alloc}\,\frac{\checkmark a}{\mathsf{emp} \vdash \Rrightarrow \exists \gamma.\, \left[\overline{a}\right]^{\gamma}} \qquad \text{update}\,\frac{a \rightsquigarrow b}{\left[\overline{a}\right]^{\gamma} \vdash \Rrightarrow \left[\overline{b}\right]^{\gamma}}$$

$$\text{where } a \rightsquigarrow b \triangleq \forall c.\, \checkmark(a \cdot c) \Rightarrow \checkmark(b \cdot c)$$

Fig. 2. Ghost state ownership

from a to b does not invalidate any possible frame. The separation logic operator $\Rrightarrow$ means "true under a frame-preserving update", and is incorporated into program logics through an extended rule of consequence:

$$\text{conseq}\,\frac{P \Rrightarrow P' \qquad \{P'\}\ c\ \{Q'\} \qquad Q' \Rrightarrow Q}{\{P\}\ c\ \{Q\}}$$

where $P \Rrightarrow Q$ is shorthand for $P \vdash \Rrightarrow Q$. In other words, we can freely perform frame-preserving updates to ghost state ("view shifts") at any point in a verification.

As an example of ghost state, consider the common proof pattern in which one party holds authoritative knowledge of the current state of a resource, and another party holds partial knowledge of one aspect of the state. For any camera R, we can define the *authoritative camera* on R as containing elements of type $R^? \times R^?$, with validity defined such that $\checkmark(a, b) \Leftrightarrow b \preceq a$. Then $\bullet a \triangleq (a, \bot)$ can be used to represent the *authoritative* value of a resource, and $\circ a \triangleq (\bot, a)$ can represent a *fragmentary* or partial value, with a guarantee that if one party knows $\bullet a$ and another party knows $\circ b$, then $b \preceq a$. This algebra allows frame-preserving updates that simultaneously modify the authoritative and fragmentary parts of the resource: for instance, if R is the algebra of key-value maps and $k \notin \mathrm{dom}(m)$, then $\bullet m \cdot \circ n \rightsquigarrow \bullet m[k \mapsto v] \cdot \circ n[k \mapsto v]$, allowing us to use a view shift to add a new key to the authoritative map and a client's knowledge of the map simultaneously.

Another common use of ghost state, and specifically higher-order ghost state, is to implement separation-logic *invariants* that are true at every step of a program's execution. The construction of invariants is somewhat complicated (see Jung et al. [2018]), but it rests on the *agreement algebra*, defined such that $\checkmark(\mathrm{agree}(a) \cdot \mathrm{agree}(b)) \Leftrightarrow a = b$. By instantiating this algebra with separation logic assertions, we can define an assertion $\boxed{I}^{\iota}$ that stores an assertion I for access by any party at any time, as long as the accessing party restores I by the end of the operation. The name ι is used to identify the invariant and ensure that it is not opened while already open; Hoare triples in Iris are indexed by a set $\mathcal{E}$ of enabled invariants, and invariants are accessed through a *mask-changing* view shift ${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}'}$,where for instance $P * \boxed{I}^{\iota}\ {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} Q$ is effectively equivalent to $P * I \Rrightarrow Q$ (i.e., the view shift ${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\{\iota\}}$ allows us to *open* the invariant ι). This kind of view shift is similarly reflected in an extended rule of consequence, where all invariants can be accessed and then restored between program steps, and is also used in the specification of *atomic* program operations, which can access invariants during their execution and restore them afterwards.

2.2 The Verified Software Toolchain

The Verified Software Toolchain (VST) [Appel et al. 2014] is, like Iris, a Coq-based separation logic verification system. It has its own generic framework for separation logic, but the vast majority of its development is specialized to one particular programming language: Clight, the first intermediate language of the CompCert verified compiler [Leroy 2009]. Clight is a syntactically restricted subset of C supporting most core language features, and CompCert's correctness proof guarantees that the assembly code generated from a Clight program has the same behavior as the source program.

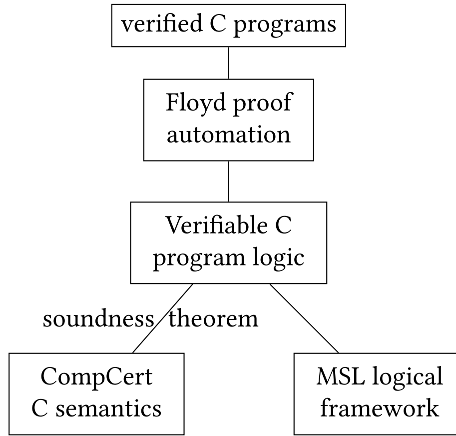

Fig. 3. The organization of the Verified Software Toolchain

VST defines a program logic for Clight, called Verifiable C, as well as an automation library for Verifiable C, called Floyd [Cao et al. 2018]. The system has been used to prove correctness of a range of real C programs in Coq, including cryptographic functions [Beringer et al. 2015], a concurrent message-passing system [Mansky et al. 2017], and a web server [Zhang et al. 2021]. Unsurprisingly, reasoning formally about real C programs is more difficult than reasoning about core calculi, and VST provides rules and automation for features like integer bounds, pointer validity, struct representation, and other details that are not present in most separation logic systems. At the same time, VST is a much older system than Iris, and while recent work has updated it with some Iris features, including limited higher-order ghost state and the Iris Proof Mode [Mansky 2022], it is still much less flexible and extensible than Iris.

In terms of the layout of VST shown in figure 3, in this work we replace VST's logical framework (MSL) with that of Iris, and then define a new resource model and definition of Hoare triples for Clight in the Iris style. We then state and prove all the Hoare rules of Verifiable C, VST's program logic, in our Iris-based logic, obtaining an Iris instance for CompCert C with the same surface rules as VST. Our new logic can be used to verify any C program previously verified with VST, but can also take full advantage of Iris's higher-order ghost state, proof mode, and other theoretical and practical innovations. Our resource model and our proofs of the rules of Verifiable C are inspired by VST, but are realized quite differently, building on ideas from Iris and taking advantage of its higher-order capabilities to do almost all our reasoning *within* the separation logic, rather than unrolling the assertions into propositions over program states. We have also mostly reconstructed VST's automation (Floyd) and used it to re-verify some simple example programs (see section 6), but due to the nature of tactic programming in Coq there are still many bugs that will only be observed in sufficiently complicated examples, and we do not present a complete automation system as part of this work.

3 A RESOURCE ALGEBRA FOR COMPCERT MEMORY

As described in section 2.1, an Iris assertion is a predicate on a collection of resource algebras, called cameras, that define the resources in the logic. Unlike other separation logic systems, Iris makes no hard distinction between "real" and "ghost" resources: all resource assertions involve ownership of some element of a camera, and it is only in the definition of Hoare triples that one particular

camera is connected to the actual physical state of the program via a *state interpretation* assertion S. By default, the physical state σ is a map from locations to values, and points-to assertions are defined via ownership of fragments of that map[1]:

$$S(\sigma) \triangleq \boxed{\bullet\sigma} \qquad\qquad \ell \mapsto_q v \triangleq \boxed{\circ[\ell := (q, v)]}$$

By the rules of the authoritative camera, $\boxed{\bullet\sigma} * \boxed{\circ[\ell := (_, v)]} \vdash \sigma(\ell) = v$, so the points-to assertions reflect the physical values held in the heap. The fraction $q \in (0, 1]$ is used to share ownership of the location (e.g., between threads): any fraction of ℓ is sufficient to load from ℓ, while full ownership is required to store to it.

CompCert complicates this picture in several ways. First, some memory locations should never be targeted by ordinary load and store operations. Any realistic model of C must include function pointers, and VST also allows designating certain locations as locks for concurrency purposes. A CompCert memory will still mark these locations as allocated, and they may even hold values, but logically they do not contain those values: instead, we associate them with special resources, LK for locks and FUN for function pointers, that do not appear in the physical memory. Second, in the standard Iris model the physical state σ does not include any ownership information: the fraction q is purely ghost state. However, CompCert memories include *permissions* at each location, chosen from Nonempty, Readable, Writable, and Freeable. The operational semantics of CompCert checks these permissions on each memory operation: a program cannot write to a location unless the location is Writable, and cannot free it unless it is Freeable. This helps ensure that optimizations verified in CompCert will continue to be sound under concurrency. There is a natural correspondence between ownership shares in separation logic and the permissions in CompCert memories. However, this correspondence points to another complication: ordinary fractional permissions do not distinguish between Writable and Freeable, and do not allow non-Readable but Nonempty ownership of a location. We must complicate our model of resource ownership to allow programs to hold enough of a location to know that it is allocated, but not enough to observe the location's value, so that other parties can write to that location but not free it.

3.1 The Camera of Shared Values

The natural algebra for representing partial ownership of resources is a pair (π, r) of a share π and resource r. By using the agreement camera for r, where $\checkmark(r_1 \cdot r_2) \Leftrightarrow r_1 = r_2$, we can guarantee that any two parties that own part of a resource agree on the current value of that resource. We can then define $(\pi_1, r_1) \cdot (\pi_2, r_2) = (\pi_1 \cdot \pi_2, r_1 \cdot r_2)$ and obtain exactly the combining and splitting rules we expect for heap resources. If we use fractions as our shares, with $\pi_1 \cdot \pi_2 \triangleq \pi_1 + \pi_2$ and $\checkmark(\pi) \triangleq \pi \in (0, 1]$, then we have that all the shares of an element total to (at most) 1, and ownership of 1 guarantees full and exclusive control over the resource. Iris uses a slightly more complicated algebra—the shares also include a *discarded* share $\square$, which represents read-only resources that can be freely duplicated and never changed—but essentially, this algebra is all we need to describe ownership of heap fragments and build a separation logic.

In this setup, the absence of 0 shares is essential. For instance, if thread A were to hold a resource $(0, v)$ while thread B held $(1, v)$, then thread B would not be able to change the value of the resource to $(1, v')$: this would be inconsistent with thread A's knowledge of the value v. In other words, changing the value of the resource is justified by holding a large enough share that no other thread can know the current value. CompCert, however, recognizes another possible distinction: it is possible for a thread to know that a location *exists* without knowing its current value, allowing

[1] More precisely, both these assertions are indexed by a fixed ghost name γ_s where state information is stored, but by convention we omit the name when there is only one relevant name in context.

other threads to write to the location without holding full ownership of it. In other words, fractional shares induce a simple lattice of ownership in which a thread may be able to read or write to a resource, with Readable < Writable; CompCert, on the other hand, uses a four-element lattice of Nonempty < Readable < Writable < Freeable. A thread with Writable but not Freeable ownership of a location can change the value of the location but cannot deallocate it; a thread with Nonempty but not Readable permission cannot read the value of the location, but can perform other operations (e.g., pointer comparison) that in C are only valid on allocated pointers. We can generalize our notion of shares to *tree shares* [Dockins et al. 2009], which naturally express this four-way distinction, but we will also have to modify the rules of our algebra to reflect the presence of *unreadable shares* that can never know the value of the resource.

The early days of concurrent separation logic saw a proliferation of mathematical models of shares/permissions for coordinating ownership of memory locations between threads. While fractions are a natural model, they are also a poor fit for some applications, especially the "token factory" pattern where a single main share splits off and recollects any number of interchangeable smaller pieces over time, while tracking the number of shares that have not yet been returned. Dockins et al. introduced tree shares [Dockins et al. 2009] as a unifying model for both fractional and token-factory shares: a share is a set of nodes drawn from an infinite binary tree, where each node can be split into its two child nodes. As applied in VST, the left subtree can be thought of as the "malloc-free share", and the right subtree can be thought of as the "read-write share". A thread that owns any nodes in the right subtree for a location ℓ can read ℓ's value, and a thread that owns the full right subtree can write to ℓ; a thread that owns any nodes in the left subtree can know that ℓ is allocated (but may not be able to read ℓ's value), and only a thread with full ownership of both subtrees can free ℓ. For our purposes, the details of tree shares are less important than the lattice they induce, in which a share may be empty; nonempty but unreadable; readable but not writable; writable but not full; or full; and any share can be split into two subshares. In fact, all the definitions in this section are agnostic in the underlying share implementation, requiring only that it satisfies this lattice; we specialize to tree shares in the next subsection.

The pair camera at the start of this section was convenient because we could combine shares and resources independently. In an algebra with unreadable shares, this is no longer the case: instead, our elements are more like dependent pairs $\{\pi \; \& \; (\text{if readable}(\pi) \text{ then } r \text{ else unit})\}$, where the second element may only be present if the first element is a readable share. Following VST, we write readable elements as $\mathsf{YES}(\pi, r)$ and unreadable elements as $\mathsf{NO}(\pi)$, and define the algebra of *shared values* as follows:

$$\mathsf{YES}(\pi_1, r_1) \cdot \mathsf{YES}(\pi_2, r_2) = \text{if readable}(\pi_1 \cdot \pi_2) \text{ then } \mathsf{YES}(\pi_1 \cdot \pi_2, r_1 \cdot r_2) \text{ else } \bot$$
$$\mathsf{YES}(\pi_1, r) \cdot \mathsf{NO}(\pi_2) = \mathsf{NO}(\pi_1) \cdot \mathsf{YES}(\pi_2, r) = \text{if readable}(\pi_1 \cdot \pi_2) \text{ then } \mathsf{YES}(\pi_1 \cdot \pi_2, r) \text{ else } \bot$$
$$\mathsf{NO}(\pi_1) \cdot \mathsf{NO}(\pi_2) = \mathsf{NO}(\pi_1 \cdot \pi_2)$$

where $\bot$ is an invalid "error" element (all non-$\bot$ elements are valid). Note that the combination of two unreadable shares is always unreadable; the combination of two readable shares, or one readable and one unreadable, may be unreadable, but only if the shares fail to combine (e.g., they add to more than 1 in the fractional case, or overlap in the tree-share case).

The algebra of shared values satisfies all the laws of cameras in Iris. It also admits the updates we desire:

LEMMA 3.1. *If π is a writable share, then* $\mathsf{YES}(\pi, r) \rightsquigarrow \mathsf{YES}(\pi, r')$.

PROOF. We must prove that $\mathsf{YES}(\pi, r) \rightsquigarrow \mathsf{YES}(\pi, r')$ is a frame-preserving update, i.e., for any c such that $\checkmark(\mathsf{YES}(\pi, r) \cdot c)$, it is also the case that $\checkmark(\mathsf{YES}(\pi, r') \cdot c)$. Let π_2 be the share of c. If

readable$(\pi \cdot \pi_2)$ holds, then π_2 must not be readable (since the combination of a writable and a readable share is $\bot$), so c must be $\mathsf{NO}(\pi_2)$. Then $\mathsf{YES}(\pi, r) \cdot c = \mathsf{YES}(\pi \cdot \pi_2, r)$, $\mathsf{YES}(\pi, r') \cdot c = \mathsf{YES}(\pi \cdot \pi_2, r')$, and both are equally valid. On the other hand, if readable$(\pi \cdot \pi_2)$ does not hold, then whether c is YES or NO we have $\mathsf{YES}(\pi, r) \cdot c = \bot$, which contradicts the assumption that $\checkmark(\mathsf{YES}(\pi, r) \cdot c)$. □

In the pair algebra we started with, only the full share 1 was sufficient to change the value of r; in the algebra of shared values, we have shown that any writable share is sufficient.

Now we can model the logical heap σ as a map from keys to shared values, which we call a *resource map* or rmap. Our definition of points-to is similar to that of Iris, but we also have an *empty points-to* assertion $\ell \mapsto _$ to represent ownership of an unreadable share of a location:

$$\ell \mapsto_\pi v \triangleq \boxed{\circ[\ell := \mathsf{YES}(\pi, v)]} \qquad\qquad \ell \mapsto_\pi _ \triangleq \boxed{\circ[\ell := \mathsf{NO}(\pi)]}$$

Ownership of both readable and unreadable points-to assertions can be split and combined according to their shares, and $\ell \mapsto_{\pi_1} v * \ell \mapsto_{\pi_2} _ \vdash \ell \mapsto_{\pi_1 \cdot \pi_2} v$, as desired. We also now have shares in the heap itself, so that the information we gain from points-to assertions is as follows:

$$\boxed{\bullet\sigma} * \ell \mapsto_\pi v \vdash \exists \pi' \geq \pi.\ \sigma(\ell) = \mathsf{YES}(\pi', v) \qquad \boxed{\bullet\sigma} * \ell \mapsto_\pi _ \vdash \exists \pi' \geq \pi.\ \mathsf{share}(\sigma(\ell)) = \pi'$$

where $\mathsf{share}(\mathsf{YES}(\pi, r)) = \mathsf{share}(\mathsf{NO}(\pi)) = \pi$. Holding $\ell \mapsto_\pi v$ guarantees that the value in the heap σ at ℓ is v, and also that the share in the heap is at least π; holding $\ell \mapsto_\pi _$ guarantees that the share in the heap at ℓ is at least π, but does not tell us what the value is or even whether a value exists ($\sigma(\ell)$ could be either YES or NO). This more complicated heap model both allows us to represent unreadable ownership of resources, and includes ownership shares in the authoritative heap.

The camera of shared values and its induced generalization of Iris's heap model are totally independent of the details of C, CompCert, or tree shares; they could be reused in any setting where we want to talk about non-readable ownership of a resource. The obvious application is to other malloc-free languages, where "writable but not freeable" is a natural permission level, but it could also be used in, e.g., a message-passing language where a thread can have enough permissions to know that a channel exists without being able to send or receive messages on it.

3.2 Coherence between Logical Heaps and CompCert Memory

Thus far, we have defined a generic algebra of heaps (rmaps) mapping locations to share-annotated resources. We will now define the type of resources we use for C, and relate our heap model to the memory model used in CompCert's semantics.

A CompCert memory [Leroy et al. 2012] consists of a set of numbered *blocks*, each of which contains a range of integer *offsets*. An *address* is a pair (b, o) of block and offset. For each allocated address, the memory tracks its current value (usually either a byte or Undef) as well as both current and maximum permissions (chosen from Nonempty, Readable, Writable, Freeable as described above). The memory also records the next block to be allocated, and guarantees that unallocated blocks have no values and no permissions. The separation of current and maximum permissions is useful for, e.g., distinguishing between read-only global variables and locations whose ownership is (temporarily) being shared.

In most Iris instances, the logical heap σ is the actual physical memory used by the operational semantics of the language. In our setting, however, a logical rmap does not uniquely determine a CompCert memory or vice versa. An rmap does not distinguish between an unallocated block and a block on which it holds no ownership, nor does it record the maximum possible ownership for a location. In the other direction, each CompCert permission except Freeable corresponds to an infinite number of possible shares: e.g., a Readable share can always be split into two smaller

Readable shares. There are also two types of values in memory, locks and function pointers, that should never be the target of CompCert memory operations (load, store, etc.), and it is useful to track these separately in our rmaps. We reconcile the two by defining a *coherence* relation between rmaps and CompCert memories, and when we state the adequacy of our logic we will reason about all memories coherent with a given rmap. First, we define the relationship between rmap shares and memory permissions:

$$\begin{aligned}
&\mathsf{perm_of}(\mathsf{YES}(\top, v)) = \mathsf{Freeable} \\
&\mathsf{perm_of}(\mathsf{YES}(\pi, v)) = \mathsf{Writable} \text{ if } \pi \text{ is writable but not } \top \\
&\mathsf{perm_of}(\mathsf{YES}(\pi, v)) = \mathsf{Readable} \text{ if } \pi \text{ is readable but not writable} \\
&\mathsf{perm_of}(\mathsf{NO}(\pi)) = \mathsf{Nonempty} \\
&\mathsf{perm_of}(\mathsf{YES}(\pi, \mathsf{LK})) = \mathsf{perm_of}(\mathsf{YES}(\pi, \mathsf{FUN})) = \mathsf{Nonempty}
\end{aligned}$$

While the logical share π is used to determine the permission for value resources, non-value resources are associated with minimal Nonempty permissions regardless of their logical ownership. This is essential for relating assertions to CompCert memories: CompCert always assigns Nonempty permissions to function-pointer locations, ensuring that they are not targeted by ordinary memory operations, but we still want to be able to reason about the contents of those locations in the logic. We describe the function-pointer assertion in section 3.3.

Next, we define the correspondence between rmaps and memories.

Definition 3.2. An rmap σ is *coherent* with a memory m when the following conditions hold:

(1) If $\sigma(\ell)$ holds a value v (i.e., not a lock or function pointer), then the value at $m(\ell)$ is v.
(2) The current permission of $m(\ell)$ is at least $\mathsf{perm_of}(\sigma(\ell))$.
(3) Blocks that have not been allocated in m are not in the domain of σ.

We could require that the current permission of ℓ be exactly $\mathsf{perm_of}(\sigma(\ell))$, but this would not be particularly useful in separation logic, since holding $\ell \mapsto_\pi v$ never precludes the existence of another share $\ell \mapsto_{\pi'} v$ (unless π is $\top$, in which case $\mathsf{perm_of}(\pi, v)$ is already Freeable).

We can now define the state interpretation assertion $S(m)$ that connects program-logic assertions to a physical CompCert memory m:

$$S(m) \triangleq \exists \sigma.\ \sigma \text{ is coherent with } m \wedge \boxed{\bullet\sigma}$$

In other words, our points-to assertions are "about" a physical m whenever the logical heap they describe is coherent with m. This allows us to derive rules in our separation logic about the interaction between points-to assertions and CompCert memory operations. For instance:

Theorem 3.3. *If* π *is* writable, *then* $S(m) * \ell \mapsto_\pi v \Rrightarrow S(\mathsf{storebyte}(m, \ell, v')) * \ell \mapsto_\pi v'$.

Proof. We know that there exists a σ coherent with m such that $\boxed{\bullet\sigma} * \ell \mapsto_\pi v$. The rules of the heap algebra give us that $\exists \pi' \geq \pi.\ \sigma(\ell) = \mathsf{YES}(\pi', v)$. Then by coherence, we know that the permission at $m(\ell)$ is at least Writable, so the storebyte operation is allowed. Next, by lemma 3.1, we know that it is a frame-preserving update to change $\sigma(\ell)$ from $\mathsf{YES}(\pi', v)$ to $\mathsf{YES}(\pi', v')$. Thus, we have $\boxed{\bullet\sigma} * \ell \mapsto_\pi v \Rrightarrow \boxed{\bullet\sigma(\ell := \mathsf{YES}(\pi', v'))} * \ell \mapsto_\pi v'$. Finally, we must show that $\sigma(\ell := \mathsf{YES}(\pi', v'))$ is once again coherent with $\mathsf{storebyte}(m, \ell, v')$. The only changed condition from definition 3.2 is condition 1, and the new value of $\sigma(\ell)$ is exactly the new value of $m(\ell)$, so coherence holds, and the proof is complete. □

In this fashion, even though our points-to assertions are not directly on CompCert memories, we can prove that operations in the logic correspond exactly to operations in the physical memory. As in Iris, but in contrast to VST, these theorems are proved entirely within our logic: we do not need

to unfold the definition of "this assertion holds on a program state" or explicitly construct heaps that satisfy specific logical assertions corresponding to memory operations. The theorems also do not yet say anything about the semantics of any programming language; in section 4.1, we will use this theorem to prove a Hoare triple like $\{\ell \mapsto_\pi v\}\ {}^*\ell = v'\ \{\ell \mapsto_\pi v'\}$ connecting the effects of program statements to the logical heap.

The approach of defining a coherence relation between the logical heap and some other notion of memory, and including this relation in the definition of the state interpretation, is a simple but general technique for separating the structure used to model the points-to assertions in an Iris instance from the memory used in the operational semantics of the underlying language. We expect it to be useful for building Iris instances for languages with more complex models of memory, especially ones that already have a defined semantics whose physical memory is not just a map from locations to values.

3.3 Locks and Function Pointers

In older separation logics (including VST's), locks and function pointers were addressed using "predicates in the heap" [Gotsman et al. 2007], where assertions (lock invariants, function specifications, etc.) were included directly in heap resources, requiring some logical tricks (e.g., step-indexing) to break the circularity between the definitions of heaps and assertions. Iris provides a far more general solution via higher-order ghost state: cameras can include predicates freely as long as their operations respect step-indexing. As outlined in section 2.1, this lets us define global invariants $\boxed{I}$ as a special case of ghost state, and then access the resources in I with a view shift. This makes it easy to define invariant-based locks without including invariants in the LK resource itself. Instead, the only argument to the LK resource is a boolean indicating whether the lock is currently acquired, and we define the lock assertion as:

$$\ell \boxdot\!\!\to R \triangleq \boxed{\exists b.\ \ell \mapsto \mathsf{LK}(b) * \text{ if } b \text{ then emp else } R}$$

This invariant captures the idea that the resources R are protected by the lock whenever the lock is not acquired[2].

We address function pointers with a similar approach. We want to define an assertion $\ell \mapsto_f \{P\}\{Q\}$ saying that ℓ points to a function with precondition P and postcondition Q. We do so by defining a camera of function specifications $\{P\}\{Q\}$, and setting up ghost state that maps each function pointer to its specification. We once again use the authoritative-fragment construction, giving us an authoritative resource $\bullet fs$ that contains the specifications of all declared functions, and a fragment $\circ[\ell := \{P\}\{Q\}]$ that knows the specification of an individual function. Then we can define the function pointer assertion as:

$$\ell \mapsto_f \{P\}\{Q\} \triangleq \ell \mapsto_\square \mathsf{FUN} * \left[\circ[\ell := \{P\}\{Q\}]\right]^{\gamma_f}$$

This assertion connects the specification $\{P\}\{Q\}$ to the FUN resource in the rmap, which in turn is connected by coherence to the location ℓ in the CompCert memory. For the share, we use the "discarded" share $\square$, which is readable and arbitrarily duplicable, and guarantees that no other thread can change or deallocate the resource at that location. This is where it becomes crucial that the perm_of function used for coherence behaves differently for different kinds of resources: in the logic, $\ell \mapsto_f \{P\}\{Q\}$ involves a readable share of ℓ, but the corresponding permission in the CompCert memory will still be the unreadable Nonempty, preventing the program from actually performing load operations on ℓ.

[2] In practice, we use a *cancellable invariant* [Jung et al. 2017] so that the lock can be deallocated before the program ends, allowing us to implement the lock reasoning rules of Hobor et al. [2008].

4 SAFETY AND ADEQUACY

4.1 Weakest Preconditions

Iris builds a program logic on top of a language semantics by defining a notion of *weakest precondition* in the logic:

$$\mathrm{wp}_{\mathcal{E}}\ e\ \{Q\} \triangleq (e \in \mathit{Val} \land \mathrel{|\!\!\Rrightarrow}_{\mathcal{E}} Q(e)) \lor \\ \forall \sigma.\ S(\sigma)\ {}^{\mathcal{E}}\!\!\Rrightarrow^{\emptyset}\ \mathrm{red}(e, \sigma) \land \triangleright \forall e_2\ \sigma_2.\ (e, \sigma) \rightarrow (e_2, \sigma_2)\ {}^{\emptyset}\!\!\Rrightarrow^{\mathcal{E}}\ S(\sigma_2) * \mathrm{wp}_{\mathcal{E}}\ e_2\ \{Q\}$$

The weakest precondition of e is the smallest assertion guaranteeing that either e is finished and satisfies the postcondition Q, or that given the state interpretation S for a physical state σ, e is not stuck and, for every (e_2, σ_2) such that (e, σ) steps to (e_2, σ_2), we can re-establish the new state interpretation $S(\sigma_2)$ and e_2's weakest precondition. The wp assertion is indexed by a set $\mathcal{E}$ of available invariant names, and the updates ${}^{\mathcal{E}}\!\!\Rrightarrow^{\emptyset}$ and ${}^{\emptyset}\!\!\Rrightarrow^{\mathcal{E}}$ allow the program to change ghost state and access the contents of invariants in each step, as long as all invariants are restored after the step. A Hoare triple $\{P\}\ e\ \{Q\}_{\mathcal{E}}$ is then defined as $P \twoheadrightarrow \mathrm{wp}_{\mathcal{E}}\ e\ \{Q\}$.

To define wp for our program logic, we need to instantiate this formula with CompCert's C semantics. The first question that arises is whether Clight's semantics fit into this framework at all. Most of the target languages for Iris are functional programming languages, with memory accesses and other operations of interest treated as side effects. Even in RefinedC [Sammler et al. 2021], function parameters and local variables are treated as bound variables that are substituted with the locations of passed parameters when called. Clight, in contrast, has an environment-based semantics, a more traditional way of describing imperative languages. A program state includes the name of the currently executing function, the next statement to execute, the memory, and two separate environments, one that maps stack-allocated variables to their addresses and one that maps temporary variables to their values. We resolve this by letting the "expression" e contain the entire program state except the memory (which is held in σ).

Beyond the expressions, there are two more semantic points that merit special attention. First, the updates ${}^{\mathcal{E}}\!\!\Rrightarrow^{\emptyset}$ and ${}^{\emptyset}\!\!\Rrightarrow^{\mathcal{E}}$ allow the semantics to access the contents of invariants in each step, as long as they are restored by the end of the step—but this is explicitly unsound in C! Suppose an invariant contained an assertion $\ell \mapsto v$: then this definition would allow two separate threads to write to ℓ without synchronizing, leading to a data race, which in C has undefined behavior (so we should not be able to prove anything about it). Existing Iris instantiations address this problem by either allowing races in the semantics (e.g., HeapLang), modifying the semantics so that race-sensitive operations occur over multiple steps (e.g., RustBelt [Jung et al. 2017]), or making the points-to assertion "subjective" so that one thread cannot use another's resources without synchronization (e.g., iGPS [Kaiser et al. 2017]/iRC11 [Dang et al. 2019]). None of these solutions are satisfying in our case: C semantics does not allow races, we want to reason about CompCert semantics directly rather than modifying it, and subjectivity is an idea from weak memory models that does not translate naturally into our sequentially consistent setting. Instead, we limit access to invariants by changing the inner masks on the updates from $\emptyset$ to $\mathcal{E}$:

$$\mathrm{wp}_{\mathcal{E}}\ e\ \{Q\} \triangleq (e \in \mathit{Val} \land \mathrel{|\!\!\Rrightarrow}_{\mathcal{E}} Q(e)) \lor \\ \forall \sigma.\ S(\sigma)\ {}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}}\ \mathrm{red}(e, \sigma) \land \triangleright \forall e_2\ \sigma_2.\ (e, \sigma) \rightarrow (e_2, \sigma_2)\ {}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}}\ S(\sigma_2) * \mathrm{wp}_{\mathcal{E}}\ e_2\ \{Q\}$$

Threads can still access the contents of invariants between program steps (e.g., by performing an opening view shift ${}^{\mathcal{E}}\!\!\Rrightarrow^{\emptyset}$ and then a closing view shift ${}^{\emptyset}\!\!\Rrightarrow^{\mathcal{E}}$ before resuming execution), but all the invariants in $\mathcal{E}$ must be enabled before we can take a program step, ensuring that we do not use resources from invariants to justify the step.

Second, CompCert's semantics includes a notion of *external function* with unknown semantics, specified by an arbitrary relation on the memories before and after the call. In VST, external calls are handled separately from the rest of the semantics: they operate on a special piece of *external state* that can only be modified by external calls. Prior work used this external state to connect to system calls provided by a verified operating system [Mansky et al. 2020], and also used external calls as a hook for a concurrency library [Cuellar et al. 2020]. We can fit external calls into the formula above by letting the state σ be a pair (m, z) of the CompCert memory m and external state z. Internal steps leave z unchanged; external steps may change both m and z according to their semantic relation. Our full state interpretation is then $S((m, z)) \triangleq S(m) * \boxed{\bullet z}^{\gamma_e}$ (where γ_e is the fixed ghost name for the external state), and the corresponding assertion $\boxed{\circ z}^{\gamma_e}$ allows the separation logic to track (but not modify) the current external state, supporting reasoning in the style of Mansky et al. [2020].

More inconveniently, external functions do not use the same form of nondeterminism as ordinary program steps. The specification for an external function *ef* has the form $\forall x.\ \{P_{ef}(x)\}\ ef()\ \{Q_{ef}(x)\}$, where the *witness* x provides logical information that does not follow directly from the state—for instance, the witness may describe the intended invariant to be used when allocating a lock. When a program reaches an external call, it may choose *any* witness x for the call such that $P_{ef}(x)$ holds, and then must be safe for *all* states satisfying $Q_{ef}(x)$. This means that in the external-function case, our definition of wp must alternate angelic nondeterminism (existential quantification) and demonic nondeterminism (universal quantification)[3]. Our final definition of wp is then:

$$
\begin{aligned}
\mathsf{wp}_{\mathcal{E}}\ e\ \{Q\} \triangleq\ & (e \in \mathit{Val} \wedge \Rrightarrow_{\mathcal{E}} Q(e)) \vee \\
& \forall \sigma.\ S(\sigma)\ {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}}\ (\mathsf{red}(e, \sigma) \wedge \triangleright \forall e_2\ \sigma_2.\ (e, \sigma) \rightarrow (e_2, \sigma_2)\ {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}}\ S(\sigma_2) * \mathsf{wp}_{\mathcal{E}}\ e_2\ \{Q\}) \vee \\
& (\exists ef\ x.\ e = (ef();\ e_2) * P_{ef}(x, \sigma) * \triangleright \forall \sigma_2.\ Q_{ef}(x, \sigma_2)\ {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}}\ S(\sigma_2) * \mathsf{wp}_{\mathcal{E}}\ e_2\ \{Q\})
\end{aligned}
$$

In the last disjunct, we existentially quantify over a witness x satisfying the precondition P_{ef}, and then must be able to re-establish $S(\sigma_2)$ and wp given any σ_2 satisfying the postcondition Q_{ef}. As we will see in the next section, our desired safety property includes this same quantifier alternation for external calls, and so this definition is sufficient to guarantee that verified programs run safely.

4.2 Adequacy

The adequacy theorem of a program logic connects theorems proved in the logic (e.g., Hoare triples) with the semantics of the underlying programming language. For our logic, we want to know that if we prove a Hoare triple for a C program, then the program will execute correctly under CompCert's semantics. Most Iris developments use or specialize a generic adequacy theorem, based on Iris's definition of wp, that says that for any program c, if we can prove $\{\mathsf{True}\}\ c\ \{v.\ Q(v)\}$ where Q is a pure assertion (i.e., it is independent of program state), then 1) it does not get stuck and 2) if it terminates in a value v, then $Q(v)$ is true. Because we have modified the definition of wp, we do not immediately obtain this adequacy theorem; furthermore, our ultimate goal is to re-prove the existing adequacy theorem for Verifiable C, which is structured to account for external calls:

Definition 4.1. A program configuration (c, σ) is *safe for n steps* with postcondition Q, where Q is a predicate on CompCert memories and external states, if:

- c is halted and $Q(\sigma)$ holds, or
- $(c, \sigma) \rightarrow (c_2, \sigma_2)$ in CompCert's semantics, and (c_2, σ_2) is safe for $n - 1$ steps, or

[3] The same issue arises in Melocoton, an in-development foreign-function-interface logic for Iris [Guéneau et al. 2023]; they provide a more general solution by transforming their step relation into a *multirelation* that alternates universal and existential quantification.

- $c = (ef();\ c_2)$ for an external function ef, and there is a witness x such that $P_{ef}(x, \sigma)$ and for all σ_2 satisfying $Q_{ef}(x, \sigma_2)$, (c_2, σ_2) is safe for $n - 1$ steps.

Theorem 4.2 (Adequacy). *For any program configuration* (c, σ), $S(\sigma) * \mathsf{wp}_\top\ c\ \{Q\} \vdash (\forall n.\ (c, \sigma)$ *is safe for n steps with* $Q)$.

Proof. By Löb induction. The definition of wp gives us three cases. In the first case, c is halted in a state satisfying Q, so it is safe for n steps trivially. In the second case, (c, σ) steps in CompCert's semantics to some (c_2, σ_2), and wp guarantees that after a view shift we have $S(\sigma_2) * \mathsf{wp}_\top\ c_2\ \{Q\}$, so safety follows from the inductive hypothesis. In the third case, c is of the form $(ef;\ c_2)$, and there is some x such that $P_{ef}(x, \sigma)$ holds; we must then show that (c_2, σ_2) is safe for $n - 1$ steps for all σ_2 satisfying $Q_{ef}(x, \sigma_2)$, which again follows from the definition of wp and the inductive hypothesis. □

As a side note, because the state σ includes external state, this definition of safety can be used to reason about the program's externally visible behavior: for instance, when c is a networked server, we can instantiate the external state with an interaction tree describing a network protocol [Koh et al. 2019; Zhang et al. 2021], and the adequacy theorem implies that the server implementation complies with the protocol.

Even though we do not reuse Iris's adequacy construction directly, the machinery of Iris still makes this proof much easier than in VST. Our proof of adequacy is 120 lines of Coq, while VST's is over 1000, involving explicit construction of rmaps corresponding to modified program states and extensive reasoning about the relationship between external function specifications and the program logic. All of this is avoided by using a separation-logic definition of wp and only exiting to meta-level reasoning after CompCert-level safety is established.

5 BUILDING THE PROGRAM LOGIC

Thus far, we have constructed a definition of weakest preconditions for CompCert C, the fundamental building block of Hoare triples. The next step is to build a program logic on top of this definition—a set of Hoare rules for the statements of Clight. We could consider reconstructing the proof rules of RefinedC [Sammler et al. 2021], or developing a new logic from scratch, but in our initial development we instead aim to reconstruct Verifiable C, the program logic of VST. Our proof rules will be syntactically identical to those of VST, but their meaning and the proofs of their soundness will be entirely different, building on the logic we have defined above rather than the original foundations of VST.

5.1 Proving the Rules of Verifiable C

As one might expect, the proof rules for C are more complicated than in most standard Hoare logics. For instance, the store rule, which might conventionally be $\{\ell \mapsto_1 v\}\ {}^*\ell = v'\ \{\ell \mapsto_1 v'\}$, appears in Verifiable C as:

$$\text{store}\ \frac{\mathsf{writable}(\pi)}{\begin{array}{c}\{(\mathsf{tc}(e_1) \land \mathsf{tc}((\mathsf{typeof}\ e_1)\ e_2)) \land (\mathsf{eval}(e_1) \mapsto_\pi v * P)\} \\ {}^*e_1 = e_2 \\ \{\mathsf{eval}(e_1) \mapsto_\pi \mathsf{eval}((\mathsf{typeof}\ e_1)\ e_2) * P\}\end{array}}$$

Aside from the fact that any writable share is sufficient to store to a location, we can see that 1) the LHS and RHS are expressions, not fully evaluated values, and 2) the rule includes typechecking conditions tc on both expressions, and, critically, these conditions are spatial assertions that may refer to memory resources. We find expressions in our rules because Clight uses a standard big-step-for-expressions, small-step-for-statements semantics, and so the program ${}^*e_1 = e_2$ does not

go through a state of the form $^*\ell = v'$ in its execution; we could potentially define an equivalent semantics that uses small steps for expressions and decompose this rule further, but here we work directly on CompCert's semantics. The inclusion of tc conditions reflects a subtle point: in CompCert, the evaluation of expressions may depend on memory. This is not because expressions can perform memory accesses—in Clight, dereferences are broken out into separate statements—but because certain operations on pointer values (e.g., pointer comparison) have undefined behavior when their operands are unallocated pointers. Verifiable C reflects this with an enhanced type system that uses separation logic assertions to guarantee that the relevant pointers are allocated: $\ell \mapsto_\pi _$ is sufficient to guarantee that ℓ is allocated, and so operations on ℓ will not get stuck. The $\mathsf{tc}(e)$ assertion collects the separation logic information needed to guarantee that e can be evaluated without getting stuck[4], after which the memory-independent $\mathsf{eval}(e)$ function can compute the value that e will return. Because the assertions in tc may overlap with $\mathsf{eval}(e_1) \mapsto_\pi v$ and may also require other memory, they are combined via a non-separating conjunction $\wedge$ with both the points-to assertion and an arbitrary frame P.

The theorem relating the memory-independent eval function to CompCert's evaluation relation $\mathsf{eval}_{\mathsf{CC}}$ is a good illustration of the contrast between the proofs of rules in VST and our new implementation of Verifiable C. The top-level theorem should say:

Theorem 5.1 (informal). *If* $\mathsf{tc}(e)$ *holds on a memory* m*, then* $\mathsf{eval}(e) = v \Leftrightarrow \mathsf{eval}_{\mathsf{CC}}(e, m, v)$.

In VST, coherence between a CompCert memory and an rmap is defined outside of the logic, and so the theorem is stated formally as:

Theorem 5.1 (VST). *If* $\mathsf{tc}(e)$ *holds on an* rmap r *coherent with a memory* m*, then* $\mathsf{eval}(e) = v \Leftrightarrow \mathsf{eval}_{\mathsf{CC}}(e, m, v)$.

In our logic, we have already embedded coherence inside our state interpretation $S(m)$, and so we can write the same theorem inside the logic:

Theorem 5.1 (new). $S(m) * \mathsf{tc}(e) \vdash \mathsf{eval}(e) = v \Leftrightarrow \mathsf{eval}_{\mathsf{CC}}(e, m, v)$.

We need not mention the rmap r; it is implicit in the model of the separation logic. The proofs are similar—in either case, we use the fact that $\ell \mapsto _$ holds on a logical heap coherent with m to derive that ℓ is allocated in m—but the new theorem statement is much easier to use inside the logic as we move forward.

This inside-the-logic approach carries forward to the proofs of the Hoare logic rules as well:

Theorem 5.2 (store). *The store rule is valid given our definition of* wp.

Proof. We must show that $Pre \vdash \mathsf{wp}\ {}^*e_1 = e_2\ Post$, where Pre and $Post$ are the pre- and postcondition of the Hoare triple for store. By the definition of wp, it suffices to show that for any state σ s.t. $({}^*e_1 = e_2, \sigma) \to (s', \sigma')$, $S(\sigma) * Pre \Rrightarrow \mathsf{wp}\ s'\ \{Post\}$. By theorem 5.1, we can conclude from $S(\sigma) * Pre$ that the calls to eval give us the same values for e_1 and $(\mathsf{typeof}\ e_1)\ e_2$ as CompCert's $\mathsf{eval}_{\mathsf{CC}}$. Therefore, the only σ' that CompCert's semantics can produce is $\mathsf{store}(\sigma, \mathsf{eval}(e_1), \mathsf{eval}((\mathsf{typeof}\ e_1)\ e_2))$. Then by theorem 3.3 we have $S(\sigma) * \mathsf{eval}(e_1) \mapsto_\pi v \Rrightarrow S(\sigma') * \mathsf{eval}(e_1) \mapsto_\pi \mathsf{eval}((\mathsf{typeof}\ e_1)\ e_2)$, completing the proof. □

5.1.1 Function Pointers and Function Calls. One particularly interesting aspect of Verifiable C is its treatment of function pointers and function specifications. In section 3.3, we defined an assertion $\ell \mapsto_f \{P\}\{Q\}$ that associates a location ℓ with a function pre- and postcondition. However, this is

[4]In contrast, RefinedC's Caesium semantics [Sammler et al. 2021] explicitly includes provenance in pointer values, which circumvents this issue but introduces complexity elsewhere in the semantics.

not enough to guarantee that if we call ℓ in a state satisfying P, it will return a state that satisfies Q. For that purpose, we need to know that every function actually satisfies its specification, which is what we are trying to prove when verifying the program! Verifiable C approaches this problem in two parts. First, we reason about every function in the context of a set of function specifications fs, and the full definition of Hoare triple includes an assumption that every function in fs satisfies its specification:

$$\mathsf{believe}(fs) \triangleq \mathop{\Huge *}_{(h,P,Q)\in fs} \triangleright \{P\}\ \mathsf{body}(h)\ \{Q\}$$

Second, we carry an assertion that connects each function-pointer assertion to the set fs:

$$\mathsf{funs_valid}(fs) \triangleq \forall \ell\ P\ Q.\ \ell \mapsto_f \{P\}\{Q\} \mathrel{-\!\!*\!\!*} (\exists h.\ (h, P, Q) \in fs * \mathsf{symb}(h) = \ell)$$

where symb is a function constructed from the program that maps each function name to its location in memory (more on this in section 5.2). This assertion tells us that every function pointer corresponds to some defined function with a specification in fs, and every function specified in fs has a corresponding function pointer in memory. (This means that every function pointer must point to a globally declared function, which would not necessarily be the case in a language with anonymous functions, but is the case in C and is explicitly required by CompCert's semantics.) Both funs_valid and believe are then included as assumptions in our definition of Hoare triples:

$$\{P\}\ c\ \{Q\}_{\mathcal{E}} \text{ with } fs \triangleq (\mathsf{believe}(fs) * \mathsf{funs_valid}(fs) * P) \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\ c\ \{Q\}$$

Every rule in the program logic is universally quantified over fs. In the top-level soundness proof, the believe assumption is discharged through Löb induction, while the funs_valid assertion is part of the precondition of the `main` function (which we discuss in detail in section 5.2).

Taken together, the believe and funs_valid assertions let us convert a function pointer assertion $\ell \mapsto_f \{P\}\{Q\}$ into the knowledge that ℓ corresponds to a defined function that satisfies $\{P\}\{Q\}$. This lets us prove the expected Hoare rule for function calls:

$$\{(\mathsf{tc}(e) \wedge \mathsf{tc}(\vec{a})) \wedge (\mathsf{eval}(e) \mapsto_f \{P\}\{Q\} * P * R)\}\ e(\vec{a})\ \{Q * R\}$$

As long as e evaluates to a function pointer with precondition P and postcondition Q, we can call it with that specification; we do not need to prove anything else about e in the call rule itself.

As a side note, RefinedC [Sammler et al. 2021] also has a function pointer assertion, but instead of describing the function's pre- and postcondition it includes the function body directly, and the user separately proves that that body satisfies a specification as needed. We have not yet done a thorough comparison of the advantages and disadvantages of these two approaches, but for now we use the specification approach for compatibility with Verifiable C.

5.1.2 Analysis. We were able to re-prove all of the rules of Verifiable C in our new logic, with their statements unchanged up to differences in notation between Iris and VST. This is encouraging: it means that everything proved in VST should also be provable in our logic. Furthermore, by doing our reasoning entirely within the program logic and taking advantage of Iris's tactics, the proofs of these rules are much simpler and easier to maintain. Discounting derived rules, Verifiable C has 14 distinct surface-level proof rules—comparable to any other program logic—but their proofs are quite complicated: in the VST development, they comprise ~12310 lines of Coq code. We prove the same rules in ~7060 LoC in our new logic, a 43% decrease.

5.2 Initial State and Whole-Program Correctness

For simple programming languages, we may use emp as the precondition of a whole program, and then allocate all the state we use inside the proofs of the relevant functions. In C, however,

some parts of the program state are initialized before the program enters the `main` function, and we must provide the corresponding assertions in the precondition of `main` for the user to be able to reason about them. There are three kinds of resources that must be present in the initial state: the memory assertions for global variables and function pointers, the table of specifications for defined functions, and a reference to the external state. The external state is just an arbitrary piece of authoritative ghost state, but the other two require more complex reasoning: we must describe an initial logical heap that is coherent with CompCert's initial memory state, and then transform it into a collection of points-to assertions for the program's global variables and function pointers.

We begin by defining a translation from CompCert memories to corresponding logical heaps. Since CompCert's permissions each correspond to a wide range of shares, we pick default shares π_w, π_r, and π_n to represent arbitrary writable, readable, and nonempty permissions. Then we can construct a logical resource for each memory location as follows:

$$\mathsf{res_of}(m, \ell) \triangleq \begin{cases} \mathsf{YES}(\top, m(\ell)) \text{ if } m(\ell) \text{ is Freeable} \\ \mathsf{YES}(\pi_w, m(\ell)) \text{ if } m(\ell) \text{ is Writable} \\ \mathsf{YES}(\pi_r, m(\ell)) \text{ if } m(\ell) \text{ is Readable} \\ \mathsf{NO}(\pi_n) \text{ if } m(\ell) \text{ is Nonempty and } m(\ell) \text{ is not a function pointer} \\ \mathsf{YES}(\square, \mathsf{FUN}) \text{ if } m(\ell) \text{ is a function pointer} \end{cases}$$

The logical heap for a memory is then simply $\mathsf{rmap_of}(m) \triangleq \lambda \ell.\ \mathsf{res_of}(m, \ell)$, which is coherent with m by construction:

Theorem 5.3. $\mathsf{rmap_of}(m)$ *is coherent with* m.

Proof. By the definition of res_of, each location with a value in $\mathsf{rmap_of}(m)$ has the same value in m, and we have chosen shares such that $\mathsf{perm_of}(\mathsf{rmap_of}(m)(\ell))$ is exactly the permission of $m(\ell)$ for each ℓ. (In particular, recall that CompCert always assigns Nonempty permission to function pointers, and $\mathsf{perm_of}(\mathsf{YES}(\pi, \mathsf{FUN})) = \mathsf{Nonempty}$.) □

This gives us a separation logic assertion that describes an initial memory m_0—namely, $\boxed{\circ\mathsf{rmap_of}(m_0)}^{\gamma_s}$—but it is not a particularly useful assertion: it monolithically describes the entire logical heap, instead of giving us ownership of each individual allocation. Our next step is to translate it into a collection of points-to assertions for the initialized memory. A CompCert C program p defines a collection of global variables $\mathsf{vars}(p)$ of the form (g, v), where g is the identifier of the global variable and v is its initial value, and a collection of functions $\mathsf{funs}(p)$; the initial memory $\mathsf{init_mem}(p)$ allocates a memory block for each of these identifiers according to a mapping symb_p from identifiers to locations.

Theorem 5.4. *Let fs be a collection of specifications for the functions in* $\mathsf{funs}(p)$. *Then*
$\boxed{\bullet\emptyset}^{\gamma_f} * \boxed{\circ\mathsf{rmap_of}(\mathsf{init_mem}(p))}^{\gamma_s} \Rrightarrow$
$\boxed{\bullet fs}^{\gamma_f} * (\mathop{\Huge *}_{(g,v)\in\mathsf{vars}(p)} \mathsf{symb}_p(g) \mapsto v) * (\mathop{\Huge *}_{h\in\mathsf{funs}(p)} \mathsf{symb}_p(h) \mapsto_f fs(h))$.

Proof. By induction on the list of definitions in p. For each defined identifier i, $\mathsf{init_mem}(p)$ initializes the block $\mathsf{symb}_p(i)$ with the appropriate data (the initial value v for a global variable, Undef for a function pointer). So $\mathsf{rmap_of}(\mathsf{init_mem}(p))(\mathsf{symb}_p(i))$ contains the resources corresponding to that data, and by splitting those resources from $\circ\mathsf{rmap_of}(\mathsf{init_mem}(p))$, we can construct the appropriate assertion, $\mathsf{symb}_p(i) \mapsto v$ for a global variable or $\mathsf{symb}_p(i) \mapsto_\square \mathsf{FUN}$ for a function pointer. Finally, in the function pointer case, we use a view shift to add the function's specification to the authoritative set of specifications and obtain a fragment assertion $\boxed{\circ[\mathsf{symb}_p(i) := fs(i)]}^{\gamma_f}$, and

then combine $\mathsf{symb}_p(i) \mapsto_\square \mathsf{FUN}$ and $\boxed{\circ[\mathsf{symb}_p(i) := fs(i)]}^{\gamma_f}$ into the function pointer assertion $\mathsf{symb}_p(i) \mapsto_f fs(i)$. □

Combining these two theorems, we can conclude that the initial memory of a program is coherent with the collection of points-to assertions for its initial global variables and function pointers. We include the global-variable assertions in the precondition for the `main` function (collected in the assertion $\mathsf{globals}(p)$), and use the function assertions to establish the funs_valid condition of section 5.1.1, guaranteeing that every specified function is present in memory and vice versa. Then in the proof of a program, we start with access to the global variables and functions of the program, and can reason about them in the same way as with memory allocated during the program. Combined with the adequacy theorem of section 4.2, this gives us the following top-level safety result:

Theorem 5.5 (Whole-Program Adequacy). *Let fs be a set of specifications for the functions in a program p, where* $fs(\texttt{main}) = \{\mathsf{globals}(p) * \boxed{z}^{\gamma_e}\}\{\mathsf{True}\}$. *Then if* $\vdash (\{P\}\, f\, \{Q\}$ *with* $fs)$ *is provable for each specification* $(f, P, Q) \in fs$, *we can conclude that* $(p, \mathsf{init_mem}(p), z)$ *is safe for any number of steps.*

Proof. By theorems 5.3 and 5.4, we can start from emp and use view shifts to allocate ghost state satisfying both the initial state interpretation $S(\mathsf{init_mem}(p), z)$ and the precondition of `main`. Instantiating the Hoare triple for `main` then gives us $S(\mathsf{init_mem}(p), z) * \mathsf{wp}_\top\ \texttt{main}\ \{\mathsf{True}\}$. By theorem 4.2 (adequacy), this implies $(\forall n.\ (p, \mathsf{init_mem}(p), z)$ is safe for n steps), as desired. □

6 VERIFYING C PROGRAMS

```
1  struct list {unsigned head; struct list *tail;};
2
3  struct list *reverse (struct list *p) {
4    struct list *w, *t, *v;
5    w = NULL;
6    v = p;
7    while (v) {
8      t = v->tail;
9      v->tail = w;
10     w = v;
11     v = t;
12   }
13   return w;
14 }
```

Fig. 4. A linked-list reverse function in C

To demonstrate the use of our logic, we reconstruct most of VST's Floyd automation library [Cao et al. 2018] on top of it, and use the Floyd tactics to verify some of VST's example programs. Figure 4 shows the code for one of these examples, a linked-list reverse function. While it is quite simple as C programs go, it does exercise C-specific features like structs and field accesses, as well as standard separation logic pointer manipulation and loop invariants. The complete proof of the

reverse function is shown in figure 5. We use semax_body to assert a Hoare triple on the Clight representation of the function, f_reverse, using a pre- and postcondition defined by reverse_spec (not shown). Vprog collects global variable information from the C program, and Gprog holds the set of defined functions and their specifications (*fs* in section 5.2).

```
Lemma body_reverse: semax_body Vprog Gprog ⊤ f_reverse reverse_spec.
Proof.
start_function.
do 2 forward.
forward_while (∃ s1 s2 w v,
  PROP (sigma = rev s1 ++ s2)
  LOCAL (temp _w w; temp _v v)
  SEP (listrep s1 w; listrep s2 v)).
* Exists (@nil val) sigma nullval p; unfold listrep at 2; entailer!.
* entailer!.
* destruct s2 as [ | h r].
  - unfold listrep at 2; Intros; subst; contradiction.
  - unfold listrep at 2; fold listrep; Intros y.
    do 4 forward.
    Exists (h::s1,r,v,y); entailer!.
    + simpl. rewrite ← app_assoc; auto.
    + unfold listrep at 3; fold listrep.
      Exists w; entailer!.
* forward.
  rewrite → (proj1 H1) by auto.
  unfold listrep at 2; rewrite → app_nil_r, rev_involutive.
  Exists w; entailer!.
Qed.
```

Fig. 5. The proof of the reverse function

The proof is not complicated, but it uses several tactics from VST's Floyd automation library: start_function to unfold the function definition and its specification, forward to symbolically execute straight-line code, forward_while to provide a loop invariant, Intros and Exists to manipulate separation logic quantifiers, and entailer! to automatically solve separation logic entailments. Our implementations of these tactics are fundamentally unchanged from VST's version of Floyd [Cao et al. 2018], but almost every supporting lemma and tactic needed at least slight changes to work with our Iris-based definitions. Once these changes were made, the VST proof script for the function body worked almost verbatim in our system: the only changes are the use of Iris's ∃ in place of VST's notation, and the addition of the invariant mask ⊤ in the arguments of semax_body, reflecting the fact that our Hoare triples are now parameterized by masks, as in Iris. We have reconstructed 13/20 of VST's basic example programs (https://github.com/PrincetonUniversity/VST/tree/master/progs64) in this way, and we expect that the rest can be reconstructed similarly—they are blocked only by various bugs in our reimplementations of the Floyd tactics, which are time-consuming but not difficult to solve. Table 1 summarizes the examples, their status, and the features of C they cover.

There are two categories of example programs whose translation to our new logic is more interesting: those that benefit from using Iris tactics in place of VST's existing automation, and

VST example	C LoC	working	features used
append.c	17	Y	linked-list struct, while loops
bin_search.c	18	Y	arrays, shift operator, while loops
bst.c	115	N	binary search tree struct, while loops
field_loadstore.c	23	Y	nested struct field access
float.c	11	Y	integer-float casts
global.c	19	Y	global variables
logical_compare.c	10	Y	boolean operators
message.c	38	N	casting between structs and byte arrays
min.c	10	N	arrays, for loop
nest2.c, nest3.c	14, 35	Y	nested structs
object.c	47	Y	object-oriented programming with method table
printf.c	48	N	printf, fprintf
ptr_cmp.c	12	Y	pointer comparison
queue.c	71	N	queue struct
revarray.c	20	N	arrays, while loops
reverse.c	34	Y	linked-list struct, while loops
strlib.c	44	N	characters, strings, for loops
sumarray.c	17	Y	arrays, while loops
switch.c	15	Y	switch statements
union.c	26	Y	unions

Table 1. VST's basic examples and their status in our system

those that used VST's prior ad-hoc ghost state implementation [Mansky 2022] and can now use Iris cameras instead. The latter examples are exclusively concurrent programs, which are out of the scope of this paper, but the former category already gives us a taste of the benefits of joining VST and Iris. For instance, VST's `object` example implements an object-oriented dynamic dispatch pattern in C using structs and function pointers: each object has a method table that contains pointers to its implementation of two methods defined in an interface. In the proof of this program, the representation predicate for the `mtable` field contains assertions about the two function pointers in the interface, whose specifications are parameterized by an underlying object implementation o:

$$\text{object_methods}(o, p) \triangleq \exists \pi\ p_{\text{reset}}\ p_{\text{twiddle}}.\ \text{readable}(\pi) \land p \mapsto_{\pi} (p_{\text{reset}}, p_{\text{twiddle}}) * \\ p_{\text{reset}} \mapsto_{f} \text{reset_spec}(o) * p_{\text{twiddle}} \mapsto_{f} \text{twiddle_spec}(o)$$

Because the share π of the `mtable` pointer is existentially quantified and function pointers are persistent, this assertion can be duplicated, i.e., $\text{object_methods}(o, p) \vdash \text{object_methods}(o, p) * \text{object_methods}(o, p)$. In VST (see the left-hand side of figure 7), proving this involves explicitly rewriting with a lemma saying that function pointers can be duplicated, and then using the automated entailment solver `entailer`. Iris, on the other hand, has built-in support for duplicating and canceling persistent assertions; an invocation of Iris's `iIntros` tactic with the `#$` pattern (# for persistence, `$` for cancelation) automatically matches one $\mapsto_f$ assertion on the left-hand side with any number on the right-hand side. When we encounter proof goals that directly manipulate separating hypotheses, manage persistent assertions, etc., we can immediately take advantage of Iris tactics that support these kinds of reasoning, simplifying our proofs.

```
1  struct object;
2
3  struct methods {
4    void (*reset) (struct object *self);
5    int (*twiddle) (struct object *self, int i);
6  };
7
8  struct object {
9    struct methods *mtable;
10 };
11
12 struct foo_object {
13   struct methods *mtable;
14   int data;
15 };
16
17
18 void foo_reset (struct object *self) {
19   ((struct foo_object *)self) -> data = 0;
20 }
21
22 int foo_twiddle (struct object *self, int i) {
23   int d = ((struct foo_object *)self)->data;
24   ((struct foo_object *)self) -> data = d+2*i;
25   return d+i;
26 }
27
28 struct methods foo_methods = {foo_reset, foo_twiddle};
```

Fig. 6. Object-oriented programming in C

7 EVALUATION

By re-implementing the logic of VST on top of Iris foundations, we obtain a new logic that

(1) can prove anything that can be proved in VST, with equally strong guarantees about the behavior of verified programs;
(2) is easier to maintain and extend than VST; and
(3) allows users to write simpler proofs in some cases by taking advantage of Iris tactics and automation.

In this section, we summarize the evidence for these assertions, give an overview of the Coq implementation of our logic, and discuss further evaluation of the system that we or others could perform.

For assertion 1, we observe that, as described in section 5.1.2, the rules of our program logic are exactly the same as VST's (up to difference in notation). This tells us that on paper, every C program that can be verified in VST can also be verified in our logic. In practice, as we have seen

```
Proof.
intros.
unfold object_methods.
Intros sh reset twiddle.
destruct (split_readable_share sh)
    as (sh1 & sh2 & ? & ? & ?); [
    assumption|].
Exists sh1 reset twiddle.
Exists sh2 reset twiddle.
rewrite ← (data_at_share_join sh1
    sh2 sh) by assumption.
rewrite (split_func_ptr' (
    reset_spec o) reset) at 1.
rewrite (split_func_ptr' (
    twiddle_spec o) twiddle) at 1.
entailer!!.
Qed.
```

```
Proof.
intros.
unfold object_methods.
Intros sh reset twiddle.
destruct (split_readable_share sh)
    as (sh1 & sh2 & ? & ? & ?); [
    assumption|].
Exists sh1 reset twiddle.
Exists sh2 reset twiddle.
rewrite ← (data_at_share_join sh1
    sh2 sh) by assumption.
iIntros "(#$ & #$ & $ & $)"; auto.
Qed.
```

Fig. 7. Proving duplicability of `object_methods` with VST (left) and Iris (right) tactics

in section 6, some VST proofs do not immediately work in our system, but this is only because of bugs in our reimplementation of the automation, and we can be sure that once we fix these bugs all prior verifications will work again. Furthermore, we prove adequacy (Theorem 4.2) against exactly the same definition of safety as VST, so the guarantees we obtain about verified programs are exactly the same as in VST. This rules out the possibility that, for instance, we have proved the same assertions about a program but subtly changed the meaning of those assertions in a way that makes the proof useless. In fact, we *have* changed the meaning of assertions by replacing their foundations, but our adequacy result shows that this does not in any way weaken the properties we prove about C programs.

For assertion 2, while we have not yet demonstrated a substantial extension of the logic (e.g., with transfinite step-indexing or later credits), we have certainly seen that it is easier to maintain. As mentioned in section 5.1.2, the soundness proofs of the logic are more than 40% smaller than in VST (we give more statistics about our Coq development in section 7.1). Those proofs are also performed at a higher level of abstraction than in VST, so they should be unaffected by minor changes in the foundations of the logic. On the foundational side, using Iris instead of an ad-hoc separation logic means that we automatically benefit from changes to base Iris: in fact, our logic already supports arbitrary higher-order ghost state and invariants without any additional effort, although these features are most easily demonstrated on concurrent programs, so we will not discuss them further here.

Finally, the `object` example in section 6 demonstrates that users can benefit from Iris Proof Mode's tactics and automation when verifying C programs in our logic. While the full power of Iris's automation comes into play primarily in concurrency reasoning, its support for named premises, persistent assertions, and Coq-style patterns is already useful in the domain of our current logic.

7.1 Coq Development

The work described in this paper is fully formalized in Coq. Our development currently uses ORA [Krebbers et al. 2018], a fork of Iris's logic that allows linear (i.e., non-garbage-collected)

resources, but this is not fundamental: while intuitively a malloc-free language should treat memory assertions linearly, it is well known that linearity alone is insufficient to prove absence of memory leaks, and the approach of Iron [Bizjak et al. 2019] can be used to prove absence of memory leaks even in an affine separation logic. Our development is structured as a branch of VST, with our contributions focused in two folders: `veric`, the Verifiable C program logic, and `floyd`, the automation library. We remove most of MSL, VST's original generic formulation of separation logic, keeping only a few standard-library lemmas and tactics, and the definition of tree shares and their associated algebra. We also add a folder `shared` that generically defines the camera of shared values from section 3.1; this folder only imports files from Iris, and could be made into a separate repository or added to Iris in the future. In Verifiable C, we add one new file instantiating the new camera and modify 44 of the remaining 75 files, reflecting the new definitions of coherence and program logic assertions, and the entirely new proofs for all Hoare rules and supporting lemmas. In Floyd, we modify 66 of 91 files, re-proving supporting lemmas and adapting tactics to the Iris implementation of Verifiable C. As described in section 5.1.2, our development of Verifiable C is significantly smaller than VST's: due to the use of Iris infrastructure and tactics and the shift from model-level to logic-level proofs, our overall `veric` development is ~44k LoC, compared to ~67k LoC in VST. Our development of Floyd, on the other hand, is roughly the same size as VST's.

7.2 Future Evaluation: Integrating Iris-Based Tools with VST

While we have demonstrated that our new system can verify the same programs in the same way as VST (and in some cases slightly more easily using Iris tactics), we have not yet evaluated the potential of our system to go beyond VST and verify new C programs in new ways. The most immediate benefit we expect is the ability to to adapt existing Iris tools and frameworks to our new Verifiable C. For instance, RefinedC [Sammler et al. 2021] is a refinement-and-ownership type system for C programs with foundational semantics in Iris, allowing users to write type annotations on C programs and semi-automatically obtain foundational correctness proofs. Reimplementing this type system on top of our logic would be a powerful demonstration of the advantages of an Iris-based VST, and would allow semi-automatic foundational verification of CompCert C programs. We do not yet know how much work will be involved in translating the definitions of RefinedC's types from its separation logic to Verifiable C—our work has bridged the foundational gaps, but there are still various differences in the interface and implementation of points-to assertions, values in memory, etc., that make the translation nontrivial.

As another example, ReLoC [Frumin et al. 2021] is a system for proving contextual refinement properties in Iris's logic, targeted to the simple functional language HeapLang. To prove that an implementation program e_i refines a specification program e_s (i.e., $e_i \preceq e_s$), ReLoC builds ghost state that represents the state of e_s in the proof of e_i, and provides proof rules and tactics for relating the steps of the two programs. By adapting the same ghost state to Verifiable C and reconstructing the proof rules and tactics, we could build a system for proving that CompCert C programs refine HeapLang programs, so that verification could be decomposed into a pass dealing with the details of C and a pass dealing with the algorithm at a higher level of abstraction. Integrating the two systems would be a good test of whether our divergences from the standard construction of an Iris logic cause problems for other Iris-based tools.

8 RELATED WORK

8.1 Program Logics for C

There are several separation-logic-based verification tools for C programs. The most obviously related is VST itself, which has been used to verify cryptographic functions [Beringer et al. 2015],

web servers [Zhang et al. 2021], and simple concurrent programs [Mansky et al. 2017]. Prior work by Mansky [Mansky 2022] integrated some features of Iris into VST, which both increased the power of the logic and highlighted the limitations of working within VST instead of moving to Iris foundations: the system could express only limited higher-order ghost state, and the proofs of program logic rules became more complicated with each feature added. We believe that building up from Iris's foundations is the more theoretically satisfying and the more practically effective approach, and hope that our work will lead to a new version of VST that is closely integrated with advances from the Iris community.

Other separation logic verifiers for C include VeriFast [Jacobs et al. 2011] and VerCors [Blom and Huisman 2014]. VeriFast in particular has integrated some Iris-style reasoning for fine-grained concurrent programs. Both systems use the common approach of reducing program correctness to a collection of first-order proof obligations that can (often) be automatically proved by SMT solvers, which makes them much easier to use but much less foundational: in fact, both systems use essentially the same program logic for both C and Java, with no underlying formal semantics for either language.

8.2 Iris Instances for Real Programming Languages

We know of four Iris instances that target some fragment of a real-world language. RustBelt [Jung et al. 2017] is built on λ_{Rust}, a core calculus for Rust that models concurrency and lifetimes. The Goose language used in Perennial [Chajed et al. 2019] automatically translates a subset of Go to a similar core calculus, with the advantage that source code can actually be written in Go (but there is still no formal connection between the translated lambda-calculus and executing Go code). Iris-Wasm [Rao et al. 2023] is the Iris instance that comes closest to giving foundational guarantees on real code, by connecting to WasmCert-Coq [Watt et al. 2021], a Coq translation of the official formal specification of WebAssembly. WasmCert-Coq has an executable interpreter, so programs verified with Iris-Wasm can actually be run.

The most directly relevant instance to our work is RefinedC [Sammler et al. 2021], which translates C to a core calculus, Caesium, based on the Cerberus reference semantics [Lau et al. 2019]. While CompCert aims to implement C more or less according to the ISO specification, Cerberus and Caesium aim to model C as it is used in writing systems code, which means that some operations (especially complicated pointer arithmetic) are undefined in CompCert but defined in Caesium; Caesium also includes some concurrency features (sequentially consistent atomic operations), which are outside the scope of CompCert. Cerberus is a partially operational model that uses SMT solvers to compute sets of allowed executions, which would make it difficult to connect to a verified compiler. The connection between Caesium and Cerberus is also not formal, so Caesium programs cannot themselves be executed. One interesting avenue unlocked by our work is to rebuild RefinedC on top of Verifiable C, enabling semi-automatic generation of foundational correctness proofs for CompCert C programs.

9 CONCLUSIONS AND FUTURE WORK

Iris has already been used to build expressive program logics with strong foundational guarantees of correctness; those guarantees are even stronger when we can connect them to verified implementations of real programming languages. We have demonstrated the construction of a program logic for CompCert C within Iris, using entirely Iris-based foundations but providing all the same surface rules and adequacy guarantees as VST/Verifiable C. CompCert presents several challenges that have not been addressed in prior Iris developments, including permissions in the physical memory, nonempty unreadable ownership, and mixed nondeterminism for external function calls, all of

which we solve here. Our logic can be used to apply all the machinery and technical innovations of Iris to prove correctness of sequential C programs compiled with CompCert.

One of the main benefits of an Iris-based CompCert logic is that it can be more easily integrated with other Iris developments. Our immediate future work is to integrate some of Iris's tools for automation, which promise to allow foundational verification to scale to much larger programs. The most directly relevant work is RefinedC [Sammler et al. 2021], which defines a separation-logic-based type system for C programs, albeit a slightly different subset of C with a slightly different semantics. RefinedC's type system is built on top of a generic framework called Lithium, and we should now be able to build a Lithium-based type system for Verifiable C, letting us semi-automatically obtain correctness proofs for CompCert C programs. Language-independent proof automation tools like Diaframe [Mulder et al. 2022] are also tempting targets for integration. Apart from automation, the ReLoC refinement proof system [Frumin et al. 2021] allows relational proofs in Iris, and could be useful both for proving security properties (e.g., information flow control) on C programs, and for refining C programs to more easily verified functional programs. A logic for CompCert C connected to the Iris ecosystem promises to vastly expand the ease of verification of real C programs and the kinds of properties we can prove.

We are also interested in the possibility of comparing the VST and Iris approaches to proof automation head-to-head. In VST, the resources available at each point in a program are treated as a "soup" that is automatically searched and modified by symbolic execution tactics, but is difficult for the user to interact with directly. Iris, on the other hand, maintains separation logic premises in a "context" similar to Coq's context of hypotheses, with names for each individual assertion and tactics for manipulating them using those names. Our system allows users to enter Iris Proof Mode and name and manipulate assertions in individual entailments, but still uses VST's top-level symbolic execution, so the Iris context is lost between steps. We would also like to develop an Iris-style tactic interface for our logic, based on existing Iris instances' `wp` tactics, where we maintain a separating context across symbolic execution steps. This would be a more familiar interface for Iris users, and would also take better advantage of Iris's support for persistent assertions, hypothesis manipulation, etc., but possibly at the cost of VST's sophisticated automation for finding and modifying assertions via symbolic execution—for instance, when executing a struct field access, VST can often automatically find the points-to assertion for the struct and modify precisely the piece of it corresponding to the field being accessed. Once we can use both sets of tactics side by side over the same logic, we can directly compare them and figure out where each might benefit from the other's techniques.

9.1 Towards a Concurrent Separation Logic for CompCert C

As mentioned in the introduction, while Iris separation logics are naturally concurrent, CompCert's semantics are strictly sequential. This means that while we could easily write a concurrent version of the separation logic presented here, it would be quite difficult to prove or even state its adequacy theorem—we would find it hard to precisely define what guarantees we get from verifying a concurrent C program. In fact, even the semantics of concurrent C programs is currently unclear: there are extensions of CompCert with specific concurrency models [Ševčík et al. 2013], and VST has an extension for well-synchronized lock-based concurrency [Cuellar et al. 2020], but neither one captures the general concurrency model of the C language. The latter framework is also both extremely complicated (involving 5 different layers of semantics and modifications to internal CompCert proofs) and unfinished: several discrepancies between CompCert and the various semantic layers have not been resolved, and the Coq development has been very difficult to maintain. We have begun work on reconstructing VST's concurrent extension, aiming to prove adequacy against the same concurrent extension of CompCert, and expect to again obtain simpler proofs by

working within the logic and taking advantage of Iris features; but in the long run, we believe that a cleaner and more generic approach is needed. Writing a concurrent separation logic for a lifted version of a sequential programming language is an interesting problem in general, and we are currently investigating techniques such as CASCompCert [Jiang et al. 2019] and DimSum [Sammler et al. 2023] that may allow cleaner decomposition of the various components of the proof.

ACKNOWLEDGMENTS

With thanks to Ralf Jung and Robbert Krebbers for in-depth discussions on CompCert algebras; to Andrew Appel, Lennart Beringer, Shengyi Wang, and Steve Zdancewic for comments on an earlier draft; and to the anonymous POPL reviewers for their feedback.

DATA AVAILABILITY STATEMENT

The artifact for this paper is available online [Anonymous 2023]. VST is on GitHub at https://github.com/PrincetonUniversity/VST/; at the time of publication, the work described is on the `vst_on_iris` branch.

REFERENCES

Anonymous. 2023. *VST on Iris*. https://doi.org/10.5281/zenodo.8423866

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press. http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB

Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 207–221.

Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing Obligations in Higher-Order Concurrent Separation Logic. *Proc. ACM Program. Lang.* 3, POPL, Article 65 (jan 2019), 30 pages. https://doi.org/10.1145/3290378

Stefan Blom and Marieke Huisman. 2014. The VerCors Tool for Verification of Concurrent Programs. In *FM (Lecture Notes in Computer Science, Vol. 8442)*. Springer, 127–131. https://link.springer.com/chapter/10.1007/978-3-319-06410-9_9

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1–4 (jun 2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying Concurrent, Crash-Safe Systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 243–258. https://doi.org/10.1145/3341301.3359632

The Coq Development Team. 2023. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.8161141

Santiago Cuellar, Nick Giannarakis, Jean-Marie Madiot, William Mansky, Lennart Beringer, Qinxiang Cao, and Andrew Appel. 2020. *Compiler Correctness for Concurrency: from concurrent separation logic to shared-memory assembly language*. Technical Report. Princeton University.

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt Meets Relaxed Memory. *Proc. ACM Program. Lang.* 4, POPL, Article 34 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371102

Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A Fresh Look at Separation Algebras and Share Accounting. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings.* 161–177. https://doi.org/10.1007/978-3-642-10672-9_13

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Log. Methods Comput. Sci.* 17, 3 (2021). https://doi.org/10.46298/lmcs-17(3:9)2021

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems* (Singapore) *(APLAS'07)*. Springer-Verlag, Berlin, Heidelberg, 19–37.

Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. (May 2023). unpublished draft.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems*

(Budapest, Hungary) *(ESOP'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 353–367. http://dl.acm.org/citation.cfm?id=1792878.1792914

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–55.

Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 111–125. https://doi.org/10.1145/3314221.3314595

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP'17: 31st European Conference on Object-Oriented Programming (LIPIcs, Vol. 74)*. 17:1–17:29.

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. ACM, New York, NY, USA, 234–248. https://doi.org/10.1145/3293880.3294106

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *Proc. ACM Program. Lang.* 2, ICFP, Article 77 (July 2018), 30 pages. https://doi.org/10.1145/3236772

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 336–365. https://doi.org/10.1007/978-3-030-44914-8_13

Stella Lau, Victor B. F. Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell. 2019. Cerberus-BMC: A Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 387–397. https://doi.org/10.1007/978-3-030-25540-4_22

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2.* Research report RR-7987. INRIA. http://hal.inria.fr/hal-00703441

William Mansky. 2022. Bringing Iris into the Verified Software Toolchain. *CoRR* abs/2207.06574 (2022). https://doi.org/10.48550/ARXIV.2207.06574

William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A Verified Messaging System. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 87 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133911

William Mansky, Wolf Honoré, and Andrew W. Appel. 2020. Connecting Higher-Order Separation Logic to a First-Order Outside World. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 428–455.

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 809–824. https://doi.org/10.1145/3519939.3523432

Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 151 (jun 2023), 25 pages. https://doi.org/10.1145/3591265

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. https://doi.org/10.1145/3453483.3454036

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-Language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 27 (jan 2023), 31 pages. https://doi.org/10.1145/3571220

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 80–95. https://doi.org/10.1145/3453483.3454031

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later Credits: Resourceful Reasoning for the Later Modality. *Proc. ACM Program. Lang.* 6, ICFP, Article 100 (aug 2022), 29 pages. https://doi.org/10.1145/3547631

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (Jun 2013), 50 pages. https://doi.org/10.1145/2487241.2487248

Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. 2021. Two Mechanisations of WebAssembly 1.0. In *FM 2021 - Formal Methods*. Beijing, China, 1–19. https://hal.science/hal-03353748

Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32

Received 2023-07-11; accepted 2023-11-07