RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types

Michael Sammler
MPI-SWS
Germany
msammler@mpi-sws.org

Rodolphe Lepigre
MPI-SWS
Germany
lepigre@mpi-sws.org

Robbert Krebbers
Radboud University Nijmegen
The Netherlands
mail@robbertkrebbers.nl

Kayvan Memarian
University of Cambridge
UK
kayvan.memarian@cl.cam.ac.uk

Derek Dreyer
MPI-SWS
Germany
dreyer@mpi-sws.org

Deepak Garg
MPI-SWS
Germany
dg@mpi-sws.org

Abstract

Given the central role that C continues to play in systems software, and the difficulty of writing safe and correct C code, it remains a grand challenge to develop effective formal methods for verifying C programs. In this paper, we propose a new approach to this problem: a type system we call RefinedC, which combines ownership types (for modular reasoning about shared state and concurrency) with refinement types (for encoding precise invariants on C data types and Hoare-style specifications for C functions).

RefinedC is both automated (requiring minimal user intervention) and foundational (producing a proof of program correctness in Coq), while at the same time handling a range of low-level programming idioms such as pointer arithmetic. In particular, following the approach of RustBelt, the soundness of the RefinedC type system is justified semantically by interpretation into the Coq-based Iris framework for higher-order concurrent separation logic. However, the typing rules of RefinedC are also designed to be encodable in a new “separation logic programming” language we call Lithium. By restricting to a carefully chosen (yet expressive) fragment of separation logic, Lithium supports predictable, automatic, goal-directed proof search without backtracking. We demonstrate the effectiveness of RefinedC on a range of representative examples of C code.

CCS Concepts: • Theory of computation → Separation logic: Automated reasoning; Type theory.

Keywords: C programming language, separation logic, ownership types, refinement types, proof automation, Iris, Coq

ACM Reference Format:


1 Introduction

Despite numerous advances in programming language technology over the past several decades, a great deal of safety- and security-critical systems software is still programmed in C. The C language remains widely used in large part because it provides fine-grained control over management of resources, which is indispensable to many systems programs. However, this control comes at the steep cost of regularly introducing serious and sometimes catastrophic bugs into code. It has thus long been one of the grand challenges of programming languages research to develop scalable formal methods that can help programmers build C code that is functionally correct, and verifiably so [2, 13, 15, 17, 19–21, 25, 27, 29, 31, 33, 40, 53, 63, 69, 75, 82, 86].

Existing tools for formal verification of C programs come in two varieties: automated or foundational.

On the one hand, automated tools like VeriFast [40], VCC [17], and MatchC [86] use a variety of techniques (including both off-the-shelf SMT solvers and bespoke separation-logic solvers) to verify correctness of C programs with minimal user intervention. With these tools, the user still needs to write specifications and provide some annotations (e.g., loop invariants) to aid the proof search, but the verification is otherwise automatic. However, automated tools have a sizable trusted computing base: one must trust that the often-sophisticated logic underpinning them is sound—and implemented correctly—since the tools do not provide any form of independently checkable proof.

On the other hand, foundational tools like VST [2, 10], as well as major verification efforts like CertiKOS [32–34] and
seL4 [51], embed expressive frameworks for verifying C code within a pre-existing logical foundation, typically a general-purpose theorem prover such as Coq or Isabelle/HOL. Foundational tools have the key advantage of a smaller trusted computing base: one need only trust the proof checker of the host theorem prover and the encoding of the operational semantics of C, but not the particular logic or implementation of the tool itself. However, the use of foundational tools typically requires significant manual proof effort: although these frameworks provide tactical support for hiding tedious proof steps, the user must still guide the proof process—e.g., manipulating the proof context, applying lemmas, performing case distinctions, unfolding definitions, instantiating quantifiers—by hand. One exception is Bedrock [13–15, 64], which provides much more powerful tactic-based automation. However, Bedrock does not handle many complexities of C, instead targeting a custom assembly-like language with a simplified memory model that prohibits many of the optimizations performed by modern C compilers [14].

In this paper, we present RefinedC, a new approach to verifying C code that is both automated and foundational, while at the same time handling a range of low-level programming idioms including pointer arithmetic, uninitialized memory, and concurrency with data races.

To support automated verification, RefinedC employs a novel type system combining refinement types and ownership types. Refinement types let us express precise invariants on C data types and strong Hoare-style specifications for C functions. Ownership types let us reason modularly about shared state and concurrency by controlling ownership of memory à la Rust [93]. Moreover, RefinedC’s type-based approach has the benefit of offering a predictable, syntax-directed approach to automated verification.

To support foundational verification, RefinedC follows the semantic typing approach of RustBelt [42, 43]: we give meaning to RefinedC’s types by interpreting them in Iris, a higher-order concurrent separation logic embedded in Coq [44, 45, 47, 55]. The typing rules of RefinedC thus simply become lemmas about our separation-logic model of types, whose soundness we establish (using Iris) in Coq. Separation logic is a natural fit for modeling RefinedC types because (a) it provides a built-in account of ownership-based reasoning, and (b) Iris provides features like invariants and ghost state, which are useful for justifying more sophisticated typing rules concerning shared state and concurrency.

**Motivating example.** Figure 1 shows a concrete example of RefinedC in action. The type `struct mem_t {` represents the state of a memory allocator: a block of memory pointed to by `buffer`, whose size is `len`. The `alloc` function tries to allocate `sz` bytes of memory from a `struct mem_t`. It first checks, using `len`, that enough memory is available, and returns `NULL` otherwise. If `buffer` is large enough, then its `last` `sz` bytes are allocated using pointer arithmetic, and `len` is updated accordingly.

```plaintext
1 struct [[rc::refined_by("a: nat")]] mem_t {
2   [[rc::field("a @ int<size_t>")] size_t len;
3   [[rc::field("&own<uninit<a>>")]] unsigned char* buffer;
4   
5   [[rc::parameters("a: nat", "n: nat", "p: loc")]]
6   [[rc::returns("(n <= a ? a - n : a) @ mem_t")]]
7   void alloc(struct mem_t* d, size_t sz) {
8     if(sz > d->len) return NULL;
9     d->len -= sz;
10    return d->buffer + d->len;
11  }
12
13 Figure 1. Memory allocator example in RefinedC.
```

The `[[rc::...]]` blocks in Figure 1 represent RefinedC annotations, which express a refined version of `mem_t` and a behavioral specification of `alloc` for RefinedC to verify automatically. Here, the refined `mem_t` is indexed by a natural number `a`, the number of bytes available from the allocator. This number must match the value stored in the `len` field as enforced using `a @ int<size_t>`, the singleton type of the `size_t` integer `a`. The `buffer` field is given the type `&own<uninit<a>>`, indicating that it is a pointer to an owned block of size `a`. Taken as a whole, the refined `mem_t` encodes the invariant that the `len` field contains the length of the owned block pointed to by the `buffer` field.

The specification for `alloc` assumes (in its `rc::args` clause) that the argument `d` points to a `struct mem_t` with `a` available bytes, and that the argument `sz` is equal to some integer value `n`. The `rc::returns` clause specifies the refined type of the value that `alloc` returns: in this case, an `optional<null>` Value, which points to an uninitialized block of length `n` if the refinement `n <= a` is true, and is `NULL` otherwise. Finally, the `rc::ensures` clause specifies that, upon returning, `alloc` gives back the ownership of `p` (the pointer passed in as the argument `d`), now pointing to a `mem_t` of the appropriately reduced size.

**Key idea.** One may wonder how the checking of richly-typed specifications like the one for `alloc` can be performed automatically. The key idea is that, even though RefinedC’s refinement types encode deep (undecidable) specifications, their syntactic structure serves to judiciously and predictably guide the proof search in a syntax-directed manner. A concrete example of this is the type `b @ optional<int,t2>` (as seen in the `rc::returns` clause in line 8 of Figure 1). Semantically, in our Iris model of RefinedC types, this type corresponds to a disjunction (untagged union) between the cases where `b` is true or false; and in general, searching for proofs of disjunctions is difficult because one may make incorrect choices, leading to backtracking. However, as we explain in §6, the

---

1Annotations use C2x attributes syntax supported by recent C compilers.
2The unrefined version `int<size_t>` is inhabited by all `size_t` integers.
syntactic structure of the program and refinement types provide crucial information that we use to make a definite choice, thus avoiding backtracking.

Formally speaking, in order to ensure that RefinedC’s typing rules lead to a non-backtracking proof search, we insist that they be expressible in a separation logic programming framework we call Lithium. Lithium is a carefully restricted fragment of the Iris logic, on which efficient goal-directed proof search is possible—indeed, we have implemented it in the form of a fully automated Coq tactic. A logic program in Lithium consists of a set of rules (often called clauses in logic programming), which serve to strategically guide proof search by instructing the Lithium interpreter how to convert every proposition into appropriate subgoals. These rules are certified correct by interpreting them semantically as lemmas to be proven in Iris (as described above). By expressing the RefinedC type system as a Lithium program, we thus obtain an automated and foundational method for checking C programs against RefinedC types, and one which is inherently extensible (e.g., to handle new C programming idioms) since it is encoded as an open set of Lithium rules.

The RefinedC toolchain. Figure 2 depicts the complete RefinedC toolchain. Developers write standard C code as they would without RefinedC. To this, they add a functional specification in the form of RefinedC’s (refinement) types and standard annotations like loop invariants. After this, RefinedC takes over. First, in step (A), a front end that we have created (based on the front end of Cerberus [66]) translates the C code to a deep embedding of C in Coq, called Caesium, and translates the annotations to RefinedC’s abstract syntax to Coq. Next, in step (B), Lithium automatically executes RefinedC’s typing rules (represented as a logic program) on the Caesium code to produce a typing derivation proving the specification in Coq. During this process, verification conditions—which are pure Coq propositions—are generated. These are mostly automatically discharged using a library of Coq tactics (step (C)), but they can also be discharged by custom (e.g., domain-specific) solvers, or manual proofs.

Under the hood, hidden from the ordinary C programmer, lie RefinedC’s types and typing rules, which have been defined ahead of time, in Lithium, by an expert. The expert must define types semantically (as explained above), and prove typing rules sound in Iris against the Caesium C semantics.

Contributions. We make the following contributions:

- RefinedC: A foundationally sound and automatic approach to functional verification of idiomatic C code based on refinement and ownership types (§4, §6).
- Lithium: A logic programming language based on the Iris separation logic, embedded in Coq, suitable for automating the type checking of RefinedC (§5).
- A front end translating annotated C code into Caesium, a deep embedding of C in Coq (§3).
- An evaluation of the RefinedC approach using case studies of varying complexity, which demonstrate RefinedC’s handling of common low-level C idioms (§7).

2 RefinedC by Example

In this section, we use motivating examples to introduce RefinedC from the user’s point of view. First, we go back in more detail to the example of Figure 1 (§2.1). We then verify the deallocation mechanism of a more complex allocator relying on a linked list of free chunks, which requires a recursive refinement type and a loop invariant (§2.2).

2.1 A Simple Memory Allocator

As shown in §1, the RefinedC annotations on struct mem_t in Figure 1 define a new RefinedC type called mem_t, which is parametric in a natural number representing the number of available bytes. We emphasize the difference between the C type struct mem_t and the RefinedC type mem_t: The C type only specifies the physical layout—e.g., the names and the offsets of the fields, which are used by the compiler to generate field accesses—but does not give meaningful correctness guarantees. For example, the C type does not enforce that len is a valid integer: it could very well be uninitialized. The RefinedC type mem_t captures the invariant satisfied by struct mem_t values on which alloc operates. Note that RefinedC specifications are purely logical: they do not influence the program’s compilation or its runtime behavior.

Specification of alloc. We now turn to the annotations assigning a type (i.e., a specification) to the alloc function. Our specification introduces a number of logical variables...
Parameters are universally quantified in the specification and, like refinements on a `struct` given with `rc::refined_by`, range over arbitrary mathematical domains (i.e., Coq types). The `alloc` function has three parameters: the natural numbers `a` and `n` representing the number of available bytes and the amount requested by the caller respectively, and the location `p` at which the allocator state is stored. These parameters connect the refinements in the argument and return types, as well as possible pre- and postconditions. The types of the arguments are specified using `rc::args` on line 7. The type `p @ down<a @ mem_t>` specifies that the first argument of `alloc` is an owned pointer to an allocator state with a available bytes, stored at location `p`. The singleton type `n @ int<size_t>` specifies that the second argument of `alloc`—the requested allocation size—is the `size_t` integer with value `n`.

Next, the return type of `alloc` is specified using `rc::returns` on line 8. The return value is an owned pointer if the allocation succeeds, otherwise it is `NULL`. These two possibilities are captured by the type `b @ optional<down<...>, null>` that represents an owned pointer if the refinement `b` is true, and `null` (the singleton type containing only `NULL`) if the refinement `b` is false. The refinement `n ≤ a` checks whether allocation will succeed (i.e., if the allocator state owns enough memory).

The last part of the specification is a postcondition marked by `rc::ensures` on line 9. It says that `alloc` returns the ownership of the `mem_t` (that it received through its first argument) back to its caller. The `mem_t` in the postcondition has an updated refinement since the amount of available memory decreases on a successful allocation. Note that the first argument of `alloc` and the type in the postcondition are refined by the same location `p`. This forces `alloc` to return ownership for the same pointer that it was passed. This ownership transfer pattern occurs often in `RefinedC`. It is inspired by Mezzo [71], and is an alternative to Rust’s mutable references.

**Verification.** `RefinedC` verifies the specification of `alloc` without manual intervention. In particular, `RefinedC`’s automation picks the correct case of the returned `optional` by examining the type of the returned value (via rules `S-NULL` and `S-OWN` on page 9). It also splits the ownership associated with `buffer` into two following the pointer addition on line 13 (via rule `O-ADD-UNINIT` on page 9). One part of this ownership stays with `buffer` while the other part is returned to the caller. In §6, we explain both techniques further, as well as how the same typing rules also automatically verify a variant of `alloc` that allocates from the start of `buffer` instead of the end.

**Error messages.** `RefinedC`’s syntax-directed proof search affords precise error messages. For example, suppose the programmer mistakenly writes `n < a` instead of `n ≤ a` in the specification of `alloc` on line 8 in Figure 1. When `n = a`, the code returns a valid pointer, while the specification expects `NULL`, causing the verification to fail:

```
Cannot solve side condition in function "alloc"!
Location: "alloc.c" [13:2-13:28]
Case distinction (n > a) → false at "alloc.c" [11:5-11:18]
... H3 : ~ n > a
c---------------------------------------
n < a
```

This error message tells the user where in the code the verification failed (at the `return` on line 13), in which branch of the `if` statement on line 11 (the else branch), and what side condition could not be proved. Using this information, the programmer can easily debug the specification.

**A thread-safe allocator.** The function `alloc` described so far cannot be used concurrently on the same `struct mem_t` object due to a data race. This is why its specification requires full ownership of the allocator state. However, `alloc` can be made thread-safe by storing its state in a global variable protected by a lock. `RefinedC` supports this through a flexible spinlock abstraction containing two abstract types, `spinlock<y>` and `spinlockedy<...>`, which are, respectively, the type of a spinlock uniquely identified by the parameter `y` and the type of values protected by the lock `y`. This interface is more general than the standard specification for locks in higher-order concurrent separation logic [38, 87] in that our `spinlocked` type allows adding resources to a lock after it has been allocated. A detailed discussion of our spinlock interface is outside the scope of this paper, but details can be found in the companion appendix [79, Section A].

### 2.2 Deallocation Using a List of Free Chunks

Next, consider the memory deallocation function `free` in Figure 3. This function inserts a chunk of memory that is being freed into a linked list of free memory chunks. When in the list, the initial bytes of a chunk are occupied by a `struct chunk`, which is a header that contains the chunk’s size (line 10), and a pointer to the next chunk (line 11) if there is one, or `NULL` otherwise. The remaining bytes of the chunk can be arbitrary.

It is an invariant of `free` that the chunk list is always sorted in increasing order of chunk size. Hence, `free` has a loop to find where to insert the new chunk (lines 27-30).

**Recursive type definition.** Figure 3 defines two C types: `chunk` of chunk headers and `chunks_t` of pointers to such headers. The type `chunks_t` (not `struct chunk`) is refined by the `RefinedC` type `chunks_t`, which is defined on line 4. The annotation `rc::ptr_type` indicates that the defined `RefinedC` type refines the type of a `pointer` to the surrounding `struct`, not the `struct` itself. The ellipsis in the definition of `chunks_t` is a placeholder for the `RefinedC` type of the struct.

Note that `chunks_t` is a recursive type: The annotation on the `next` field mentions `chunks_t` again. Unfolding of recursive types is handled by `RefinedC` automatically; no extra annotations are required to indicate when to unfold.
typedef struct {
  [rc::refined_by("s: {gmultiset nat}")]
  [rc::ptr_type("chunks_t")]
  [rc::exists("n: nat", "tail: {gmultiset nat}")]
  [rc::size("n")]
  [rc::constraints("(s = {{[n]} \uplus tail})",
                  "(\forall k, k < n \rightarrow n \leq k)")]
} chunk {
  [rc::field("n @ int\<size\>\<t\>")]
  [rc::size_t size;]
  [rc::field("tail @ chunks_t")]
  [rc::refined_by("s: {gmultiset nat}")]
  [rc::parameters("m: multiset nat", "n: nat")]
  struct chunks_t;
  struct chunks_t entry = data;
}

chunks_t* cur = list;
while((cur != NULL)
   if(sz <= (*cur)->size)
   cur = (*cur)->next;
   break;
}

Figure 3. Example of an allocator with a freelist.

**Multiset and invariant.** We explain the type chunks_t further. This type is refined by a multiset of natural numbers to ensure that the chunk contains the sizes of all chunks in the list. When chunks_t is an owned pointer (i.e., when s is not the empty set), the struct that points to it is parameterized by the size of the first chunk n and the multiset tail refining the rest of the list. These two parameters are existentially quantified in the rest of the type (rc::exists annotation). A constraint (rc::constraints annotation) relates n and tail to s. A second constraint says that n is less than or equal to all elements of tail, which implies that the list of chunks is sorted. The last interesting point about chunks_t is the rc::size annotation on line 6. This annotation means that the chunk actually occupies n bytes in memory, of which the C type (struct chunk) only describes the initial part. In other words, the chunk is of size n bytes and a struct chunk (the header) is overlaid at its beginning. The remaining bytes of the chunk are treated as uninitialized by RefinedC.

**Loop invariant and verification.** The formal specification of free should be unsurprising. It says that when free is passed a free list with chunks of sizes s and a pointer to an owned chunk of size n (this is the block to be freed), then at the end of free, the free list contains chunks of sizes ([n]) \uplus s (using Coq multiset operation notations). Importantly, free has a precondition (line 17) that the block being added to the free list is large enough to fit the struct chunk header.

Verifying free in RefinedC requires an explicit loop invariant (lines 22-26). Loop invariants are described with up to three annotations: rc::exists introduces local, existentially quantified logical variables, rc::inv_vars specifies RefinedC types of relevant program variables at the start of each loop iteration, and rc::constraints lists additional assertions. (This example does not need rc::constraints.)

The loop invariant tracks the ownership of the list as it is traversed. Logically, the list has two parts: the suffix that has not yet been traversed and the prefix that has already been traversed. These two parts are pointed to by the local variable cur and the argument variable list, respectively. The loop invariant associates ownership of the list’s two parts to these two variables. Specifically, it introduces a multiset variable s corresponding to the multiset refinement of the suffix and asserts that cur points to an owned list of chunk sizes from cs. Next, it asserts that if this ownership extended with a chunk of size n (the new chunk) is combined with the ownership associated with list, then one obtains ownership of the entire output list (sizes from multiset ([n]) \uplus s). This if-then relation is conveniently expressed using the wand<...> type using a standard technique for expressing partial data structures via the magic wand of separation logic.

Finally, the annotation rc::tactics on line 19 instructs RefinedC to use the multiset solver from the std++ Coq library [92] for proving the side conditions in this example that RefinedC’s default solver cannot prove.

# RefinedC Front End and Caesium

Before a C program can be verified by RefinedC, it is elaborated by the RefinedC front end to a core language we call Caesium. This language is control-flow graph-based, and given a formal semantics through a deep embedding in Coq. The core of this semantics is a low-level memory model that is roughly based on that of CompCert [61, 62]. Caesium provides both sequentially consistent and non-atomic memory accesses, and assigns undefined behavior to data races following the semantics of RustBelt [42]. Caesium supports many low-level idioms like pointer arithmetic, the address-of operator (also on local variables), access to representation bytes, fixed-size integers, goto (including unstructured switches, such as Duff’s device), alignment checks, composite types as arguments and return values, uninitialized memory with poison semantics [59], and first-class function pointers. The RefinedC front end is implemented in OCaml and relies on the first half of the pipeline of Cerberus [66].
practices commonly found in low-level systems code disagree on many aspects of the C memory model [66, 67, 103]. Hence, the Caesium memory model has less undefined behavior than ISO C with respect to, e.g., padding in structs and effective types.

Caesium lacks some features of ISO C that are subject to active research. It does not support C’s loose expression evaluation ordering [29, 37, 52] (Caesium fixes a left-to-right ordering), lifetimes of block-scoped variables [37, 57] (all local variables are function-scoped in Caesium), integer-pointer casts [49, 66], and relaxed-memory concurrency [5, 22, 48] (Caesium’s only atomic accesses are sequentially consistent). To mitigate the first two points, the RefinedC front end performs an over-approximating analysis that emits warnings if an expression may be non-deterministic, or if the address of a block-scoped variable could escape.

**Trusted computing base.** The trusted computing base (TCB) of RefinedC includes the implementation of the front end, the definition of the Caesium semantics, and Coq. The front end contains around 6000 lines of OCaml code (excluding Cerberus) that transform Cerberus’s AIL intermediate language into a control-flow graph and translate AIL constructs to Caesium (almost 1-to-1). The definition of the Caesium semantics is currently roughly 1500 lines of Coq code (including some proofs) and additionally uses definitions from the Coq standard library, std+++, and the language interface of Iris. The Iris logic itself is not part of the TCB since its adequacy theorem establishes a closed Coq statement that involves just the operational semantics. Similarly, the Lithium interpreter described in §5 need not be trusted since it generates proofs in Iris.

### 4 RefinedC Types and Specifications

This section describes RefinedC’s types further. Several interesting RefinedC types, along with their intuitive meaning, are shown in Table 1. (These types also appeared in earlier examples.) In RefinedC, most types can have a refinement, an optional parameter that limits values in the type. A refinement is a logical predicate on values of the type, but the meta-level sort of the refinement and the predicate vary from type to type. For example, the type $n \in \text{int}(a)$ can be refined by a mathematical integer $n$ to form the type $n \in \text{int}(a)$ that represents the singleton set $\{n\}$ of $a$-sized integers. The type $\phi \in \text{bool}$ is the single Boolean value reflecting the validity of proposition $\phi$. The refinement type $\ell \in \&\text{own}(\tau)$ denotes an owned (non-aliased) pointer and its refinement $\ell$ specifies the exact memory location that is owned. As examples, the annotations on $\text{mem}_t$ on line 3 in Figure 1 use $\&\text{own}(\tau)$ to denote a pointer to a block of $n$ bytes of uninitialized memory. The type $\phi \in \text{optional}(\tau_1, \tau_2)$ is a type-level case distinction on the validity of $\phi$. It is most commonly used to represent nullable pointers (via $\&\text{own}(\tau)$ and null), as illustrated in §2. Another interesting type is $\text{wand}(H, \tau)$, which is used to encode partial data structures via the magic wand [11]. The loop invariant of $\text{free}$ in Figure 3 uses this type.

The last four types in Table 1 are most often generated from other annotations (although they can be used directly, too). A structure type $\forall \sigma \exists \tau \psi$ is built by combining the types given by the $\text{rc::field}$ annotations on a C structure (e.g., lines 2-3 in Figure 1). The types $\exists x. \tau(x)$ and $\{\tau | \phi\}$ are generated from $\text{rc::exists}$ and $\text{rc::constraints}$ annotations (e.g., lines 5-7 of Figure 3). Finally, the type padded$(\tau, n)$, which represents type $\tau$ padded to $n$ bytes, is generated from $\text{rc::size}$ annotations (e.g., line 6 of Figure 3).

**Function types.** Functions have RefinedC types of the form $\text{fn}(\forall x. \tau_{\text{arg}}; H_{\text{pre}}) \rightarrow \exists y. \tau_{\text{ret}}; H_{\text{post}}$. Function types are generated from the source code annotations we have already seen. For example, the annotations on $\text{alloc}$ (lines 6-9 of Figure 1) lead to the function type $\text{alloc}_{\text{spec}}$ shown in Figure 4. Logical variables in the $\text{rc::parameters}$ annotation (line 6) correspond to $x$ in the function type, the annotations $\text{rc::args}$ and $\text{rc::returns}$ (lines 7-8) correspond to $\tau_{\text{arg}}$ and $\tau_{\text{ret}}$, respectively, and the annotations $\text{rc::requires}$ and $\text{rc::ensures}$ (line 9) correspond to $H_{\text{pre}}$ and $H_{\text{post}}$, respectively. Existential variables that are bound in the return type and the postconditions by $\text{rc::exists}$ correspond to $y$. RefinedC function types are first-class: functions can be stored in memory and passed to or returned from other functions.

RefinedC assigns types to C programs through a type system consisting of several typing judgments and typing rules. Before introducing these judgments and rules, we describe the fragment of the Iris separation logic in which RefinedC’s typing rules are represented in Coq.

### 5 Lithium: Separation Logic Programming

RefinedC’s typing rules lie in a fragment of the Iris separation logic for which proof search can be directed entirely by the goal to be proven, without backtracking. This enables us to automate RefinedC efficiently. In this section, we define
We note that reasoning logic connectives allocate in the typing assumptions about function state judgemen\(\vdash\)tionically, it is equivalent to goal \(\Delta \vdash G\). Lithium is simpler and never backtracks.

**Lithium syntax.** A Lithium judgment has the form \(\Gamma; \Delta \vdash G\), where \(G\) is the goal to be proven, and \(\Gamma\) and \(\Delta\) are two contexts of hypotheses whose elements can be used as an arbitrary number of times (unrestricted) and at most once (restricted), respectively. The syntax of contexts and goals is:

- **Atom** \(A \ ::= \ell \cdot \tau \mid v \cdot \sigma \mid \ldots\)
- **Basic goal** \(F \ ::= \Sigma_{\text{stmt}} s \mid A_1 <: A_2 \{G\} \mid \ldots\)
- **Goal** \(G \ ::= \text{True} \mid F \mid H \ast G \mid H \rightarrow G \mid G_1 \land G_2 \mid \forall x. G(x) \mid \exists x. G(x)\)
- **Left-goal** \(H \ ::= \Gamma[\phi] \mid A \mid H \ast H \mid \exists x. H(x)\)
- **Contexts** \(\Gamma \ ::= \emptyset \mid \Gamma, x \mid \Gamma, \phi \mid \Delta := \emptyset \mid \Delta, A\)

The unrestrict ed context \(\Gamma\) contains universally quantified variables (parameters) \(x\) and pure propositions \(\phi\), all of which are duplicable. The resource context \(\Delta\) contains atoms \(A\). The atom \(\ell \cdot \tau\) expresses that location \(\ell\) has type \(\tau\), and the atom \(v \cdot \sigma\) expresses that value \(v\) has type \(\sigma\). Atoms are nonduplicable because types may contain resource ownership.

Next, we describe goals, \(G\). The simplest goals are **basic goals**, denoted \(F\). Basic goals represent RefinedC typing and subsumption (subtyping) judgments. For example, the basic goal \(A_1 <: A_2 \{G\}\) is a RefinedC subsumption judgment; logically, it is equivalent to \(A_1 \Rightarrow (A_2 \ast G)\). The basic goal/typing judgment \(\Sigma_{\text{stmt}} s\) means that the C statement \(s\) is well-formed in the function state \(\Sigma\), which contains the control-flow graph and the postcondition of the function containing \(s\).

As an example, we show below the Lithium judgment stating that alloc has the type in Figure 4. Importantly, RefinedC typing assumptions about alloc’s arguments are represented in the Lithium context (to the left of \(\vdash\)), and \(\Sigma\) contains the postcondition of alloc, i.e., the consequent of alloc spec.

\[
\begin{align*}
\emptyset; \ell_0 \cdot \sigma_1 \cdot p \& \&own(a \@ \text{mem}_t), &
\ell_2 \cdot n \@ \text{int(size}_t) \& \& own(\text{uninit}(a)) \\
\end{align*}
\]

Besides basic goals, goals \(G\) may also contain the separation logic connectives \(\ast, \ast, \land, \forall, \text{ and } \exists\). However, the left sides of \(\land\) and \(\ast\) are restricted to a smaller class of goals called left goals, \(H\), which cannot contain \(\land, \forall\), and \(\ast\). We explain the exact purpose of this restriction later but, briefly, it significantly narrows the search space for proofs.

**Goal-directed search.** The search for a proof of \(\Gamma; \Delta \vdash G\) in Lithium is directed by the goal \(G\), and proceeds by case analysis of \(G\). We summarize the cases below. The action in each case is based on standard introduction and rewriting rules of separation logic.

1. \(G = \text{True}\): The search succeeds trivially.
2. \(G = G_1 \land G_2\): Fork to prove both \(\Gamma; \Delta \vdash G_1\) and \(\Gamma; \Delta \vdash G_2\).
3. \(G = \forall x. G'(x)\): Prove \(\Gamma, y; \Delta \vdash G'(y)\) for a fresh \(y\).
4. \(G = \exists x. G'(x)\): Prove \(\Gamma; \Delta \vdash G'(x)\) for a fresh evar \(x\).
5. \(G = F\): Find a RefinedC typing rule \(\frac{G'}{G}\) whose conclusion \(F'\) can be unified with \(F\), and prove \(\Gamma; \Delta \vdash G'\).
6. \(a. G = (H_1 \ast H_2) \ast G'\): Prove the equivalent judgment \(\Gamma; \Delta \vdash H_1 \ast G\) and \(\Gamma; \Delta \vdash H_2 \ast G'\); the next step will analyze the smaller formula \(H_1\).
7. \(a. G = \exists x. (H(x) \ast G)\): Prove the equivalent Lithium judgment \(\Gamma; \Delta \vdash \exists x. (H(x) \ast G)\) and use case (4); the next step will analyze a smaller formula \(H(x)\).

**No backtracking.** The Lithium proof search procedure is efficient in large part because it does not backtrack. Several design choices make this possible. First, the left side of \(\ast\) in goals is limited to the form \(H\), which cannot contain \(\land, \forall, \text{ and } \ast\). Without this restriction, proving a goal \(G_1 \ast G_2\) would require a two-way split of the resource context \(\Delta\) to prove \(G_1\) and \(G_2\) simultaneously, requiring backtracking over possible splits of \(\Delta\). However, when \(G_1\) is limited to the form \(H\), we can reduce it in place all the way down to atoms (case (6) and its subcases), which eliminates this form of backtracking.

Second, the left side of \(\ast\) in goals is also restricted to the form \(H\). This allows us to reduce local assumptions to atoms before adding them to the context \(\Delta\) (case (7) and its subcases). By keeping only atoms in \(\Delta\), we eliminate backtracking over possible hypotheses that can be used to prove a given goal atom of the form \(\ell \cdot \tau \text{ or } v \cdot \sigma \cdot \tau\). We trivially

\[
\begin{align*}
\text{alloc spec} & \triangleq \text{fn}(\forall (\ell, n, p). p \& \& own(a \@ \text{mem}_t), \ n \@ \text{int(size}_t)\text{; True}) \\
& \quad \rightarrow \exists(). \ (n \leq a) \@ \text{optional}(\& \& own(\text{uninit}(a)), \text{null}); p \& (n \leq a) \text{? (a} - n \text{) : a} \@ \text{mem}_t \\
\end{align*}
\]
match ℓ or v from the goal to each hypothesis and at most one hypothesis will match, since the context ∆ won’t contain multiple typing assumptions for the same location or value. In principle, backtracking could arise in case (5), where more than one RefinedC typing rule could match the goal F. However, multiple matches do not actually arise because RefinedC’s typing rules are syntax-directed: types and code inside F uniquely determine the applicable typing rule.\footnote{Lithium also offers a way to specify priority among RefinedC rules in case this property fails to hold. But once a rule is chosen, RefinedC does not backtrack on the choice.}

Handling of evars. One important aspect of Lithium not mentioned so far is the handling of evars created in case (4). In particular, Lithium must be careful when instantiating evars because a bad instantiation could easily make the goal unprovable. To prevent this, case (4) seals the evars it creates so that they cannot be prematurely instantiated by Coq’s unification. In fact, the only place sealed evars can get instantiated is when solving side conditions emitted by case (6c), at which point Lithium attempts to eliminate any evars in the side condition using one of the following heuristics.

First, Lithium tries to find a suitable instantiation for the evars by checking if the side condition is an equality and, if so, removing the seals from all evars and then trying to unify both sides (potentially instantiating evars). Though this heuristic is often effective, it may also turn a provable goal into an unprovable goal if it unifies an evar appearing as the argument of a non-injective symbol. For example, unifying (length ?x) and (length l) will lead to ?x being instantiated with l, whereas the correct instantiation for ?x might in fact be another list with the same length as l. In such cases, the user’s only recourse at present is to adjust the annotations to generate side conditions in an order that allows correct instantiation. However, this has not caused problems in the examples we have tried so far. In particular, all examples of §7 use this heuristic.

Second, if Lithium cannot instantiate the evars in the side condition, it simplifies the goal using a set of user-extensible rewriting rules and equivalences. For example, a side condition of the form ?xs ≠ [] is simplified to the equivalent ∃y. ∃ys. ?xs = y :: ys, which leads Lithium to introduce evars ?y and ?ys and instantiate xs with ?y :: ?ys. The simplification rules are also used by case (7c) to normalize assumptions introduced into the context. For example, an assumption xs + ys = [] is simplified to xs = [] and ys = [], which causes both xs and ys to be substituted with []. By default, this simplification mechanism applies equivalences and thus preserves provability, but there is an escape hatch that lets one add implications (rather than equivalences) as simplification rules. (Doing so can make provable goals unprovable.)

The procedure described above is not complete as there can be a side condition for which the heuristic for evar instantiation fails and no simplification rule applies. However, the predictable nature of goal-directed search in Lithium helps the user avoid such side conditions: since Lithium always processes goals from left to right, it is straightforward to predict in which order the side conditions will be generated. For example, when checking a function call the arguments (rc::args) are checked before additional preconditions (rc::requires), so one need not worry about evars in the preconditions if they are determined by the arguments. (See free in Figure 3 for an example of this.) Finally, for the uncommon case where the above heuristics fail, the user has two fallback options: they can either extend the simplification rules or choose RefinedC annotations more carefully in order to generate simpler unification problems.

Implementation. We have implemented a Lithium interpreter in the Ltac language \cite{coq} of Coq. The interpreter maps Γ to the standard Coq context and ∆ to the spatial context provided by the Iris Proof Mode \cite{iris1,iris2}. The search for matching RefinedC typing rules (case (5) above) is handled using Coq’s typeclass mechanism \cite{coq84}. For unification, we leverage Coq’s unification. The simplification mechanism for side conditions containing evars is based on a combination of the automatch tactics and typeclasses.

Extensibility. Inspired by the semantic typing approach of RustBelt \cite{rustbeltr, rustbeltr2}, RefinedC types and typing judgments are defined semantically in terms of the connectives of the Iris separation logic, and typing rules are proved as lemmas in Iris. This means that RefinedC can be extended with user-defined types and typing rules. RefinedC’s extensibility is reflected in Lithium’s automated proof search as well: when new typing rules are added, Lithium’s proof search automatically uses them through case (5) above.

6 Examples of RefinedC Typing Rules

Next, we explain selected typing rules, shown in Figure 5. Every typing rule has the form \( \frac{\Gamma}{G \vdash F} \) where \( G \) is a Lithium goal and \( F \) is a Lithium basic goal, which encodes a RefinedC typing judgment.

Judgment basics. RefinedC has a specialized typing judgment for each program construct, e.g., \( \text{\texttt{rc::requires}} \) for conditional statements and \( \text{\texttt{rc::binop}} \) for binary operators. These judgments are parameterized by the types of the values they operate on. This ensures that Lithium’s proof search does not need to backtrack since these types uniquely determine the applicable rule. For example, consider the rules If-bool and If-int in Figure 5. Depending on the type of the condition (bool vs. int) a different rule applies and typing proceeds differently. Such type-based overloading allows RefinedC to handle the same program construct differently depending on the context. This is useful because, in C, the same construct may serve different purposes.

Construct-specific judgments arise in the premises of rules for general statement and expression judgments, e.g., T-if or T-binop. The expression judgment \( \tau \text{-}\text{expr} e \{ v, \tau. G(v, \tau) \} \),
whose premise requires typing the boolean expression which also appears in the premises of rules, is a bit unusual
since it is parameterized by a continuation \( G \), similar to the
postcondition of the weakest precondition assertion in Iris.
This continuation has two purposes. First, typing an expression
`infers` a type \( \tau \). This type, together with an inferred
(symbolic) value \( v \) for the result, is passed as an
argument to the continuation. Second, the continuation is used to
linearize type checking as in \( \text{T-BINOP} \), which first types \( e_1 \), then,
in the continuation, types \( e_2 \), and, after both \( \tau_1 \) and \( \tau_2 \) have been inferred, introduces \( \tau_{\text{T-BINOP}} \). This continuation-passing
style ensures that every typing rule’s premise has one logical
formula, which simplifies Lithium’s implementation.\(^6\)

**Typing rules for optional.** As demonstrated in §2.1, the
optional type of RefinedC plays a key role in handling the
common low-level programming pattern of encoding an
error value as \( \text{NULL} \). Most uses of this pattern can be handled
by three RefinedC typing rules: the rule \( \text{O-optional-eq} \) for
comparing an optional with \( \text{NULL} \), and the two rules \( \text{S-null} \)
and \( \text{S-own} \) for introducing an optional type.

The rule \( \text{O-optional-eq} \) is used to prove \( \tau_{\text{STATM}} \) \( \text{if } e \text{ then } s_1 \text{ else } s_2 \). To do this, Lithium first applies \( \text{T-IF} \),
whose premise requires typing the boolean expression \( e = \text{NULL} \). It then applies \( \text{T-BINOP} \), which requires typing \( e \). Suppose Lithium infers the type \( \phi @ \text{optional}(\&\text{own}(\tau), \text{null}) \)
for \( e \). Next, Lithium types the second expression, \( \text{NULL} \). This is trivial as \( \text{NULL} \) has type null. At this point, Lithium’s goal
is a judgment that matches the conclusion of \( \text{O-optional-eq} \).

We now explain \( \text{O-optional-eq} \) in detail. The rule distinguishes
two cases via \( \land \), corresponding to the cases where \( \phi \)
holds or does not hold. When \( \phi \) holds (first case), \( v_1 \) must be
an owned pointer, which cannot equal \( \text{NULL} \), so the result
of the equality check in the conclusion of the rule must be false.
Accordingly, in this case, the continuation \( G \) is checked with
argument \( \text{false} \), and \( \phi \) and \( v_1 \&\&\& \text{own}(\tau) \) are added to the context (using case (7c) and case (7d) of §5). When \( \phi \) does not
hold (second case), \( v_1 \) must have the type null, so \( v_1 \) must be
\( \text{NULL} \) and, hence, equal to \( v_2 \). Accordingly, the continuation \( G \)
is checked with argument \( \text{true} \) and \( \neg \phi \) added to the context.

In either of these two cases, the typing of \( \tau_{\text{STATM}} \) \( \text{if } e = \text{NULL} \) then \( s_1 \) else \( s_2 \) continues using \( \text{IF-BOOL} \) (with the meta-
variable \( \phi \) of \( \text{IF-BOOL} \) instantiated to False or True, respectively).
This rule also distinguishes two cases, but one holds
vacuously by virtue of the new assumption False (or \( \neg \text{True} \)).
Next, we explain how Lithium establishes that a value \( v \)
have type \( \phi @ \text{optional}(\&\text{own}(\tau), \text{null}) \). A typing goal is an atom (A) in Lithium, so the proof starts with case (6d) of §5.
Accordingly, Lithium looks for an atom \( A' \) in the context that

\[^6\]In the companion appendix we list all RefinedC judgments [79, Section B]
and give more typing rules for \( \tau_{\text{STATM}} \) and \( \tau_{\text{EXPRT}} \) [79, Section C].
types v. Typically, A’ will type v at either null or &own(r’) for some r’. In the first case, (6d) yields a new goal of the form v < s, null << v ⪰ (ϕ @ optional(&own(r), null)) {G’} (for some continuation G’). At this point, rule S-NULL is used to reduce the goal to proving ¬ϕ (and G’), which is what one expects from the intuitive meaning of the optional type. In the second case, Lithium’s goal is v < s, &own(r’) << v ⪰ (ϕ @ optional(&own(r), null)) {G’}. Using rule S-OWN, this reduces to proving ϕ and a subsumption between r’ and r, which again follows the meaning of the optional type.

Ownership reasoning. Next, we explain how program syntax guides ownership reasoning in RefinedC. Consider the expression d->buffer + d->len on line 13 of Figure 1. Logically, this expression splits the ownership of d->buffer into two parts: one part that remains associated with d->buffer, and a second part that is returned to the caller with the allocated memory. This reasoning is performed by the rule O-ADD-UNIT, which types the addition of an integer n2 to a pointer to uninitialized memory of length n1 (RefinedC type uninit(n1)). The rule splits uninit(n1) into the smaller pieces uninit(n2) and uninit(n1 − n2), after checking that n2 ≤ n1. This rule is a representative instance of how RefinedC’s informative types disambiguate the intended logical meaning of a commonly overloaded C operator (+ in this case).

Note that O-ADD-UNIT can be reused in other contexts where programs add values of type &own(uninit(n)) and int(size_t t). For example, say we change the implementation of a1loc to allocate from the beginning of buffer instead of the end, i.e., replacing line 13 in Figure 1 with the following:

```c
1  unsigned char *res = d->buffer;
2  d->buffer += s2;
3  return res;
```

RefinedC automatically verifies the resulting version of a1loc without further changes since O-ADD-UNIT is general enough to cover the type checking of + in both cases. The only difference is that the two versions distribute v1 and v1 + t n2 differently. In the original version, v1 and the associated &own(uninit(n2)) stay in buffer, while v1 + t1 n2 is returned with &own(uninit(n1 − n2)). In the new version, buffer is updated to v1 + t1 n2, while the original value v1 is returned.

Fine-grained concurrency. RefinedC can also automatically verify fine-grained concurrent code. We illustrate this with the type atomicbool(H_v, H_u), which represents a Boolean that can be accessed atomically. The type holds the ownership of H_v if the Boolean is true, and of H_u if the Boolean is false. For example, a spinlock that protects the resource H can be modeled as the type atomicbool(True, H).

The main atomic operation supported by the atomicbool type is atomic_compare_exchange_strong, corresponding to Cesium’s CAS(ℓ_atom, ℓ_exp, v<cls) operation. The first argument (ℓ_atom) is a pointer to the current expected value of ℓ_atom, and the third argument (v<cls) is the value to be assigned to ℓ_atom. CAS also sets ℓ_exp to the previous value stored at ℓ_atom.

CAS is verified using the rule Cas-bool. The second and third arguments of CAS have singleton Boolean types that determine whether the premise uses H_v or H_u. CAS-bool has two cases corresponding to whether the CAS fails or succeeds. (First case) When CAS fails, the second argument is updated to ¬b1, and false is returned. (Second case) When CAS succeeds, we receive ownership stored with the atomic Boolean before the CAS, and have to prove ownership stored after the CAS. Subsequently, we receive ownership of v0, and the CAS returns true. (The implementation of the spinlock mentioned earlier uses Cas-bool with b1 ⊆ false and b2 ⊆ true, which means that on a successful CAS, one receives the ownership of H stored in the spinlock.)

The RefinedC type atomicbool hides complex Iris concepts related to fine-grained concurrency like impredicative invariants and ghost state. These concepts show up only in proving the soundness of Cas-bool, which we have done once and for all in Coq. Lithium’s automation only uses the much simpler statement of the Cas-bool rule, not its proof.

7 Evaluation and Case Studies
To evaluate the automation and expressiveness of RefinedC, we verified full functional correctness of six classes of programs in Table 2. We selected these programs to cover a wide variety of reasoning patterns ranging over standard benchmarks (#1), tricky ownership reasoning (#2), difficult side conditions (#3, #4), real-world C code (#5) and concurrent algorithms (#6).

First, Table 2 lists the most interesting types used by each example. This shows how RefinedC types like wand or padded are reused across different programs. Then, the table shows the number of RefinedC typing rules used in type checking each of the examples. All typing rules used by the examples are either automatically generated unfolding rules for user-defined types or they are part of the RefinedC standard library. This standard library currently contains around 30 types and 200 typing rules. As explained in §5, Lithium automatically selects and applies the right typing rule from these predefined rules. Table 2 shows how many such automatic rule applications Lithium performs. This number gives a sense of the automation afforded by Lithium, showing the extent to which typing rules handle tasks like ownership manipulation and unfolding of definitions that must be performed manually in some other tools. Additionally, the table shows how many existential variables are automatically instantiated via the heuristics described in §5. Across all programs, we had to instantiate only one evar manually (in Spinlock).
Table 2. Evaluation of RefinedC. Types used: Salient type constructs used in the program. Rules: Number of distinct typing rules / number of typing rule applications. ∃: Number of automatically instantiated existential quantifiers. \( \forall \phi \land \): Number of side conditions automatically proved / manually proved. Impl: Lines of C code (counted by tokei [94]). Spec: Lines of top-level (function) specification. Annot: Lines of annotation in source code (numbers in parentheses show breakdown into data structure invariants / loop annotations / other annotations). Pure: Lines of pure Coq reasoning, including definitions and lemma statements. Ovh: Sum of Annot and Pure divided by Impl.

<table>
<thead>
<tr>
<th>Class</th>
<th>Test</th>
<th>Types used</th>
<th>Rules</th>
<th>( \exists )</th>
<th>( \forall \phi \land )</th>
<th>Impl</th>
<th>Spec</th>
<th>Annot</th>
<th>Pure</th>
<th>Ovh</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Singly linked list</td>
<td>wand, alloc</td>
<td>44/613</td>
<td>119</td>
<td>47/5</td>
<td>106</td>
<td>33</td>
<td>24</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Queue</td>
<td>list segments, alloc</td>
<td>42/310</td>
<td>81</td>
<td>10/0</td>
<td>42</td>
<td>15</td>
<td>9</td>
<td>0</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>Binary search</td>
<td>arrays, func. ptr.</td>
<td>40/308</td>
<td>68</td>
<td>73/6</td>
<td>42</td>
<td>16</td>
<td>6</td>
<td>0.5</td>
<td>19</td>
</tr>
<tr>
<td>#2</td>
<td>Thread-safe allocator</td>
<td>wand, padded, lock</td>
<td>58/319</td>
<td>96</td>
<td>28/2</td>
<td>68</td>
<td>18</td>
<td>21</td>
<td>14/2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Page allocator</td>
<td>padded</td>
<td>40/236</td>
<td>60</td>
<td>14/0</td>
<td>43</td>
<td>14</td>
<td>14</td>
<td>0.5</td>
<td>0.6</td>
</tr>
<tr>
<td>#3</td>
<td>Bin. search tree (layered)</td>
<td>wand, alloc</td>
<td>50/964</td>
<td>216</td>
<td>50/11</td>
<td>133</td>
<td>65</td>
<td>22</td>
<td>8/7</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>Bin. search tree (direct)</td>
<td>wand, alloc</td>
<td>48/977</td>
<td>240</td>
<td>47/43</td>
<td>115</td>
<td>43</td>
<td>17</td>
<td>8/7</td>
<td>0.2</td>
</tr>
<tr>
<td>#4</td>
<td>Linear probing hashmap</td>
<td>unions, arrays, alloc</td>
<td>57/1167</td>
<td>356</td>
<td>175/39</td>
<td>111</td>
<td>46</td>
<td>34</td>
<td>14/7</td>
<td>2.7</td>
</tr>
<tr>
<td>#5</td>
<td>Hafnium mpool allocator</td>
<td>wand, padded, lock</td>
<td>72/1730</td>
<td>515</td>
<td>122/11</td>
<td>191</td>
<td>53</td>
<td>55</td>
<td>28/19</td>
<td>0.3</td>
</tr>
<tr>
<td>#6</td>
<td>Spinlock</td>
<td>atomic Boolean</td>
<td>25/65</td>
<td>10</td>
<td>14/1</td>
<td>24</td>
<td>12</td>
<td>13</td>
<td>0/1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>One-time barrier</td>
<td>atomic Boolean</td>
<td>18/34</td>
<td>5</td>
<td>6/0</td>
<td>20</td>
<td>7</td>
<td>2</td>
<td>0/2</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 2 also lists how many pure side conditions RefinedC solves automatically using its default solver and how many need at least some manual help. We count these numbers very conservatively: In many cases, a standard solver, like set_solver from std++ [92], discharges several side conditions automatically, but we still count these side conditions in “manual” since the developer has to explicitly specify that the set solver must be used. Basically, any side condition that cannot be discharged by the one default solver that we wrote—which currently only targets linear arithmetic and Coq lists—is counted as manual. This default solver can definitely improve in the future. Finally, for each example, Table 2 lists the number of lines of C code, annotations, and pure Coq reasoning for manual proofs. Importantly, there is no column for the number of lines of separation logic (Iris) reasoning since the RefinedC automation is able to handle this automatically (with the exception of the initialization function for spinlocks, which we explain later).

Overall, our experience is that RefinedC’s automation can handle a wide variety of low-level reasoning, requiring manual input only for example-specific pure (mathematical) side conditions and only in the more challenging examples. RefinedC’s relative annotation overhead is moderate—less than 0.7 for all examples that do not involve complex side conditions (which are not the focus of RefinedC’s automation at present).

#1: Common case studies. The first three examples of Table 2 are case studies common to many verification tools. The verification of singly-linked lists uses the representation of partial data structures with magic wand [11, 12] illustrated in §2.2, while the verification of queues needs a more specialized notion of list segments. Both use the first allocator of #2 below for the allocation of new nodes. The five side conditions counted here as manually discharged are actually handled automatically by set_solver from std++. Additionally, we verified a binary search implementation using a function pointer, and a client of it. RefinedC handles this easily since function pointer types are first class. The annotation overhead for these examples is low. In addition to annotations for loops and data structure invariants, only a single annotation (to import manual proofs) is necessary.

#2: Ownership reasoning. To evaluate RefinedC’s ownership reasoning, we verified two memory allocators. These examples showcase RefinedC’s expressiveness, as all necessary ownership transfers can be represented using types like padded (rc::size annotation in Figure 3). The thread-safe allocator uses annotations to manipulate the spinlocked type, as described in §2 and in the companion appendix [79, Section A]. (A third memory allocator from real-world code is covered in #5 below.)

#3: Layered vs. direct verification. A popular approach to verification of low-level code is to split the verification tasks into many layers of intermediate specifications [34, 63]. To investigate how this layered approach works in RefinedC, we verified a binary search tree first via an intermediate functional layer, and second by directly going from C to the desired specification as a functional set. Although both approaches are viable with RefinedC, the overhead of the
direct approach is significantly less than the overhead of the layered approach as it does not require defining the intermediate layer. The direct approach works well because the type system cleanly separates ownership reasoning from pure functional reasoning and all except three side conditions are automatically discharged by variants of set_solver.

4: Complex functional reasoning. To check whether RefinedC scales to data structures with complex functional invariants, we verified a hashmap with linear probing. Verifying linear probing is non-trivial since all keys share the same array, and one has to prove that an insertion or deletion does not affect unrelated keys. The verification uses a functional version of the probing function for stating the invariant. RefinedC reduces verification to pure reasoning about this invariant, which is discharged through manual proofs in Coq.

5: Real-world code. Our largest case study applies RefinedC to a version of the page allocator8 of the Hafnium hypervisor [35]. This verification combines many of the previously mentioned techniques, and shows that RefinedC can verify real-world C code. Even though this allocator is significantly more complicated than the allocators in #2, we did not have to define any new RefinedC types to automatically handle the ownership reasoning.

6: Concurrent abstractions. The examples in this class show that RefinedC can automatically verify fine-grained concurrent code that is out of reach for many other automatic verifiers. In particular, we use the atomic Boolean type from §6 to verify two concurrent algorithms: a spinlock and a one-time barrier. This type is abstract enough to automate the verification of the acquire and release functions of the spinlock and the barrier. The initialization function needs manual proofs where it allocates a ghost token and for instantiating one existential quantifier with a newly generated ghost name. As mentioned in §2, RefinedC also provides a spinlocked type, which decouples the spinlock from the resources protected by it; the typing rules for spinlocked require 162 lines of additional Iris proofs. Altogether, the result is a reusable spinlock abstraction, which is used by several other examples in Table 2 (the first allocator of #2, and the allocator of #5).

8 Related Work

Bedrock. Like RefinedC, the Bedrock project [13–15, 64] targets foundational and mostly automatic separation logic-based verification of low-level programs. However, Bedrock is based on a custom assembly-like language and custom DSLs built on top, using macros that are verified similar to compiler passes [14, 15]. In contrast, RefinedC applies to existing C code that can be compiled using off-the-shelf optimizing C compilers.

Another point of difference from RefinedC is that, rather than exploiting the higher-level abstractions of a refined type system to drive automation, Bedrock encodes specifications and abstract predicates in plain separation logic, for which proof automation [13, 64] can be extended via custom Ltac tactics and hints for unfolding abstract predicates. However, Bedrock’s hint format is less expressive than Lithium, e.g., it cannot represent rules like Cas-bool from §6. Also, unlike RefinedC typing rules, Bedrock hints cannot be tied to specific program constructs and, hence, cannot be directed by program syntax. Thus, for example, the verification of a singly-linked list requires four custom hints and ~10 lines of custom Ltac in Bedrock [91], whereas no such extra work is required in RefinedC. (Both tools require loop invariant annotations.)

VST. VST [2, 10] is a separation logic-based framework for verifying CompCert C programs. Users of VST deploy a set of semi-automatic tactics to build functional correctness proofs in Coq [10], or a front end [102] that uses source code annotation to reduce verification to a set of entailments that have to be proven in Coq. However, in both cases the user needs to manually guide the proof by performing case distinctions, applying lemmas, unfolding predicates, and instantiating existential quantifiers—tasks that RefinedC’s Lithium-based automation handles automatically in most cases. As a concrete example, verification of a binary search tree similar to the one in §7 by the authors of VST [97] requires manual effort for hundreds of such proof steps, which is not the case in RefinedC. (The binary tree example in RefinedC needs manual effort only for pure side conditions.)

Foundational verification of large-scale C programs. There are several projects that perform C verification at scale, most notably seL4 [51] and CertiKOS [32–34].

seL4 [50, 51] demonstrated the first formal proof of functional correctness of a complete, general-purpose operating-system kernel and comes with a translation-validation procedure [68, 81] to transfer the proofs to generated assembly code. However, most of seL4’s proofs about C code are manual and rely only on basic tactic support [50, 104]. Later work automates some but not all of the most tedious parts [30, 31]. This automation, and the original seL4 verification, do not support some aspects of C (such as concurrency and taking addresses of local variables) that are supported by RefinedC.

CertiKOS [32–34] provides the first correctness proof of a general-purpose concurrent OS kernel with fine-grained locking. CertiKOS verification is integrated with the CompCert C compiler, so the proof applies to the generated assembly code. The proof technique used (called “certified abstraction layers”) is based on writing programs at different layers of abstraction and proving refinements between these layers. Refinement proofs are discharged (broadly similar to VST) by manually guiding specialized tactics in Coq. As seen in §7, RefinedC does not (in most cases) require such
manual guidance in Coq, and it also supports a layer-based approach (although quite different from CertiKOS’s, since RefinedC’s is based on layers of types vs. layers of programs in CertiKOS). However, further work is needed in order to establish the effectiveness of RefinedC at the larger scale at which seL4 and CertiKOS have been deployed.

Non-foundational tools for verification of C. We compare RefinedC to some of the most closely related non-foundational tools for verifying C code.

VCC [17] employs SMT solvers to verify C programs and has been used on large C programs in practice. However, it lacks good support for dynamic ownership reasoning. For example, a linked list predicate that supports member testing requires three ghost fields—all of which need to be updated manually in the add function [18, 95]. No such ghost fields and annotations are necessary in RefinedC.

VeriFast [40] is an automated, separation logic-based verification tool for C and Java. It provides heuristics to automatically infer annotations to reduce the proof burden [100]. VeriFast’s symbolic execution approach (of which only a core subset has been proven sound [99]) uses a fixed rule for each program construct, whereas RefinedC allows type-based overloading as described in §6. RefinedC also benefits from existing Coq libraries like std++ [92]: the binary search tree (layered) example from §7 requires roughly half the number of lines of pure reasoning compared to a similar example in VeriFast [96] by judicious use of existing lemmas and tactics. Other than this, the annotation burden is similar.

MatchC [78, 86] is an automated verification tool for C based on the K framework [77] and matching logic [76]. Its rewrite-based approach provides good automation for non-trivial pointer-manipulating programs and can be extended with new abstractions and custom rules like RefinedC. However, unlike RefinedC, these abstractions and their rules are not proven sound against a model, and must be trusted. MatchC also does not support concurrency.

Verification of crypto. Various projects have embedded subsets of C suitable for crypto verification in off-the-shelf verification tools. Fiat Crypto [26] provides a language for crypto in Coq, which is compiled to C. Fiat Crypto is used to verify a high-performance implementation of the P-256 elliptic curve. Low* [72] provides a semi-foundational approach to C verification through a shallow embedding of C in F∗ [88], which is then extracted to C. Verification of Low* code can use the full power of F∗, including SMT. Low* is used in the verified HACL∗ cryptographic library [107]. Due to the exclusive focus on crypto, these projects do not support some features of C that are supported by RefinedC, such as recursive data types, function pointers, and concurrency.

Separation logic automation. The verification literature abounds in (non-foundational) automatic solvers for separation logic and frame inference [16, 58, 60, 70, 73, 90]. These solvers are usually specialized for a certain class of atomic formulas (usually a variant of the symbolic heap fragment [6] of separation logic), rely on more sophisticated automation (e.g., based on SMT solvers), and can automate more difficult reasoning patterns (e.g., induction reasoning [16]) than Lithium. In contrast, proof search in Lithium is conceptually more straightforward (which makes it more predictable and amenable to implementation in a proof assistant), and has no built-in knowledge about atoms and atomic formulas; rather, it relies on the user to extend it with domain-specific atoms and typing rules. This makes Lithium extensible with custom abstractions and adaptable to many different reasoning patterns used in idiomatic C code.

Logic programming languages for linear and separation logic. Prior work on logic programming for linear or separation logic [1, 3, 36, 39] focuses on identifying large subsets of the underlying logic that remain amenable to logic programming. However, these fragments need expensive techniques like backtracking. In contrast, Lithium is deliberately limited to the smallest subset of separation logic that suffices for a type system. Proof search in a type system is directed by program syntax and types, and typically does not require backtracking. Accordingly, we eliminate backtracking from Lithium, which makes it easier to implement a certifying interpreter for it in Coq.

Memory safety in low-level programming languages. RefinedC focuses on full functional verification of low-level programs. Much prior work [7, 9, 106] focuses instead on the different—and simpler—problem of automatically verifying memory safety. One popular approach [20, 25, 69] is to combine static and dynamic checks to enforce safety of C programs. In contrast, RefinedC targets verification without affecting the dynamic semantics of the program. Low-Level Liquid Types [75] verify memory safety of C code using a combination of refinement types [74] and alias types [83, 101]. The annotation overhead is low (e.g., no loop invariants are required), but the goal is only memory safety. In contrast, RefinedC targets full functional verification, and thus requires more annotations but can also verify more programs (e.g., it addresses the limitations described by Rondon et al. [75, Section 5.1]). Finally, safety can also be attained by using a memory-safe language such as Vault [24], Cyclone [41, 89], or Rust [93] in place of C. However, these languages rely on runtime checks, and—unlike RefinedC—cannot guarantee functional correctness.

Refinement and ownership type systems. Refinement types [28, 74, 105], although originally developed for functional programs, have also been used for the safety and correctness of imperative code [4, 75, 98]. This line of work usually focuses on fully automatic type systems for relatively simple imperative languages. In contrast, RefinedC requires more annotations (e.g., loop invariants), but can verify more complicated properties and supports a more realistic subset of C (including pointer arithmetic, uninitialized
memory, and concurrency). A recent, closely related piece of work in this area is ConSORT [98], which, like RefinedC, combines refinement types with ownership types. ConSORT achieves a higher degree of automation by using a simpler model of ownership types. However, ConSORT does not support abstractions like the magic wand and atomic Booleans that are used in many of the programs in §7.

**Foundational verification of fine-grained concurrent algorithms.** There is an abundance of related work on foundational verification of fine-grained concurrent algorithms using interactive proofs, e.g., in FCSL [80], VST [65], and Iris [46, 56]. This line of work has focused on more challenging concurrent algorithms than the spinlock and barrier we have verified in RefinedC. In future work, we aim to investigate if we can develop types besides the atomic Boolean type (§6) that would enable automatic verification of more sophisticated concurrent algorithms.

**Semantic typing.** RefinedC’s semantic typing approach—in particular, building a semantic model of types on top of Iris—is modeled after that of RustBelt [42]. However, the concrete design of RefinedC’s type system differs from RustBelt in several key aspects: (1) RefinedC uses Mezzo-like [71] alias types [83, 101] instead of Rust’s lifetimes and mutable references, (2) RefinedC includes refinement types in addition to ownership types, and (3) RefinedC supports automated type checking, which RustBelt does not.

### 9 Limitations and Future Work

In this paper, we have demonstrated the potential of refined ownership types to effectively automate the foundational verification of C code. However, RefinedC is still in its infancy and has a number of limitations that we plan to address in future work.

**C idioms and features.** RefinedC relies on an expert crafting typing rules to handle relevant programming idioms in the code one wishes to verify. Our evaluation shows that it is possible to come up with reusable typing rules for several common C programming idioms. However, there are C programming idioms that are not yet covered by our existing typing rules. For example, although array accesses are already well-supported, good typing rules for pointers to array elements remain to be developed.

Also, Caesium and the front end lack support for some features of C like floats or integer-pointer casts. The former is mostly a matter of engineering since mechanized libraries for floating point semantics exist [8], while the latter requires more research to find the right semantics [49, 66].

Furthermore, RefinedC relies on syntactic, not semantic, equality of pointers (see Lithium’s case (6d)). This suffices in many cases because RefinedC is designed so that computing the same pointer twice (e.g., taking the address of the same field twice) results in the same pointer syntactically. However, some C code, in particular code using integer-pointer casts (not currently handled by RefinedC), requires a proper treatment of pointers that are semantically equal, but syntactically unequal. One idea would be to use a solver for semantic equality of locations in case (6d).

RefinedC currently does not support reasoning about external function calls and input-output behavior of programs. We believe that the automation provided by Lithium can also be useful for such I/O verification.

**Pure automation and evars.** So far, we have focused mainly on automating the separation logic aspects of reasoning. We additionally support automation for several domains of pure reasoning by leveraging existing solvers for e.g., linear arithmetic, sets, and multisets, but this support can certainly be extended further. Furthermore, as described in §5, the handling of evars is known to be incomplete in certain cases and can be improved.

**Liveness properties.** RefinedC only verifies partial, not total, correctness. This is mainly due to Iris’s focus on verifying safety properties. However, recent work enables termination verification in Iris using transfinite step-indexing [85]. It would be interesting to combine transfinite step-indexing with RefinedC and Lithium to achieve automated and foundational verification of liveness properties.

### Acknowledgments

We thank Ralf Jung and Jan-Oliver Kaiser for many useful discussions, Ike Mulder, Paul Zhu, and Emanuele D’Osualdo for their early adoption of RefinedC, and our shepherd Tahina Ramananandro and the anonymous reviewers for their helpful feedback. Additionally, we thank Peter Sewell, Hong-Seok Kim, Chung-Kil Hur, Neel Krishnaswami, Christopher Pulte, Jean Pichon-Pharabod, Jeung Kim, Youngju Song, Ben Laurie, Sarah de Haas, the pKVM development team, and all other people involved in the pKVM verification effort for their useful feedback and support.

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project “RustBelt”, funded under the European Union’s Horizon 2020 Framework Programme (grant agreement no. 683289), in part by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (AdG grant agreement No 789108, ELVER), in part by the Dutch Research Council (NWO), project 016.Veni.192.259, in part by the EPSRC Programme Grant REMS: Rigorous Engineering of Mainstream Systems (EP/K008528/1), in part by a Google PhD Fellowship for the first author, and in part by generous awards from Android Security’s ASPIRE program and from Google Research.

### References
