RefinedC: A Foundational Refinement Type System for C Based on Separation Logic Programming

Michael Sammler
MPI-SWS

Rodolphe Lepigre
MPI-SWS

Robbert Krebbers
Radboud University Nijmegen

Kayvan Memarian
University of Cambridge

Derek Dreyer
MPI-SWS

Deepak Garg
MPI-SWS

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 still 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.

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 programming applications. 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, 12, 14, 16, 18–20, 24, 26, 28, 30, 32, 39, 52, 62, 68, 74, 82, 85].

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

On the one hand, automated tools like VeriFast [39], VCC [16], and MatchC [85] 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 code 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, 9], as well as major verification efforts like CertiKOS [31–33] and seL4 [50], 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 code 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 [12–14, 63], which provides much more powerful tactic-based automation; but it 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 [13].

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 [92]. 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 [41, 42]: we give meaning to RefinedC's types by interpreting them into Iris, a higher-order concurrent separation logic embedded in Coq [43, 44, 46, 54]. 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.

The `[[rc::...]]` blocks in Figure 1 represent RefinedC annotations:[1] these serve to 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`.[2] The `buffer` field is given the type `&own<uninit<a>>`, indicating that it is a pointer to an *owned* block of memory 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` 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` with the appropriate residual 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,

```
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
6  [[rc::parameters("a: nat", "n: nat", "p: loc")]]
7  [[rc::args   ("p @ &own<a @ mem_t>", "n @ int<size_t>")]]
8  [[rc::returns("{n ≤ a} @ optional<&own<uninit<n>>, null>")]]
9  [[rc::ensures("p @ &own<{n ≤ a ? a - n : a} @ mem_t>")]]
10 void* alloc(struct mem_t* d, size_t sz) {
11   if(sz > d->len) return NULL;
12   d->len -= sz;
13   return d->buffer + d->len;
14 }
```

Figure 1. Memory allocator example in RefinedC.

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<T1,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 is forced to make potentially incorrect choices, leading to backtracking. However, as we explain in §6, the *syntactic* structure of the program and refinement types provide crucial information that we use to make a definite choice, thus *avoiding backtracking altogether*.

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.[3] 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 [65]) translates

[1] Annotations use C2x attributes syntax supported by recent C compilers.
[2] The unrefined version `int<size_t>` is inhabited by all `size_t` integers.
[3] The implementation of RefinedC (together with case studies) is provided as a companion artifact [79].

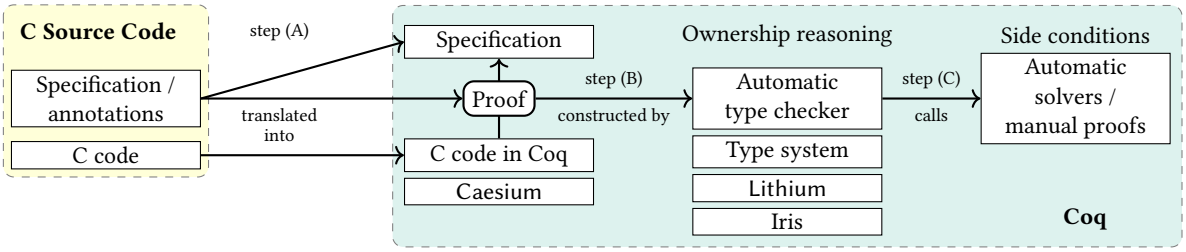

Figure 2. The architecture of RefinedC.

the C code to a deep embedding of C in Coq, called **Caesium**, and translates the annotations to RefinedC's abstract syntax in 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 `a` 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 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 (`rc::parameters` on line 6). 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 @ &own<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<&own<...>, 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`[4] 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

[4]Curly braces `{...}` are used to delimit Coq code in RefinedC annotations.

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 [70], 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 right case of the returned `optional` by examining the type of the returned value (via rules S-NULL and S-OWN on page 8). It also splits the ownership associated with `buffer` into two following the pointer addition on line 13 (via rule O-ADD-UNINIT on page 8). One part of this ownership stays with `buffer` while the other part is returned to the caller. §6 explains both techniques further.

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 sidecondition 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
----------------------------------------
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<γ>` and `spinlocked<γ, ...>`, which are, respectively, the type of a spinlock uniquely identified by the parameter γ and the type of values protected by the lock γ. This interface is more general than the standard specification for locks in higher-order concurrent separation logic [37, 86] as 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 Sammler et al. [78, 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.

```
1  typedef struct
2  [[rc::refined_by("s: {gmultiset nat}")]]
3  [[rc::ptr_type("chunks_t:"
4                 "{s ≠ ∅} @ optional<&own<...>, null>")]]
5  [[rc::exists  ("n: nat", "tail: {gmultiset nat}")]]
6  [[rc::size    ("n")]]
7  [[rc::constraints("{s = {[n]} ⊎ tail}",
8                    "{∀ k: nat, k ∈ tail → n ≤ k}")]]
9  chunk {
10   [[rc::field("n @ int<size_t>")]] size_t size;
11   [[rc::field("tail @ chunks_t")]] struct chunk* next;
12 }* chunks_t;
13
14 [[rc::parameters("s: {gmultiset nat}", "p: loc", "n: nat")]]
15 [[rc::args      ("p @ &own<s @ chunks_t>", "&own<uninit<n>>",
16                  "n @ int<size_t>")]]
17 [[rc::requires  ("{sizeof(struct_chunk) ≤ n}")]]
18 [[rc::ensures   ("p @ &own<{{[n]} ⊎ s} @ chunks_t>")]]
19 [[rc::tactics   ("all: multiset_solver.")]]
20 void free(chunks_t* list, void* data, size_t sz) {
21   chunks_t* cur = list;
22   [[rc::exists  ("cp: loc", "cs: {gmultiset nat}")]]
23   [[rc::inv_vars("cur: cp @ &own<cs @ chunks_t>")]]
24   [[rc::inv_vars("list:"
25        "p @ &own<wand<{cp ◁ₗ ({[n]} ⊎ cs) @ chunks_t},"
26                     "{{[n]} ⊎ s} @ chunks_t>>")]]
27   while(*cur != NULL) {
28     if(sz <= (*cur)->size) break;
29     cur = &(*cur)->next;
30   }
31   chunks_t entry = data;
32   entry->size = sz; entry->next = *cur;
33   *cur = entry;
34 }
```

Figure 3. Example of an allocator with a freelist.

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: `struct 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.

Multiset and invariant. We explain the type `chunks_t` further. `chunks_t` is refined by a multiset of natural numbers `s` on line 2. This multiset 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 it points to 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]} ⊎ 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 `cs` 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]} ⊎ 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 [10].

Finally, the annotation `rc::tactics` on line 19 instructs RefinedC to use the multiset solver from the std++ [91] Coq library for proving the side conditions in this example, as RefinedC's default solver cannot prove them.

3 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 [60, 61]. Caesium provides both sequentially consistent and non-atomic memory accesses, and assigns undefined behavior to data races following the semantics of RustBelt [41]. 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 [58], and first-class function pointers. The RefinedC front end is implemented in OCaml and relies on the first half of the pipeline of Cerberus [65].

Since RefinedC aims at the verification of low-level systems code (like allocators, as shown in §2), the Caesium semantics is more permissive than what the ISO C standard describes. Indeed, it is well documented that ISO C and de facto practices commonly found in low-level systems code disagree on many aspects of the C memory model [65, 66, 102]. 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 [28, 36, 51] (Caesium fixes a left-to-right ordering), lifetimes of block-scoped variables [36, 56] (all local variables are function scoped in Caesium), integer-pointer casts [48, 65], and relaxed-memory concurrency [5, 21, 47] (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.

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 $\mathsf{int}(\alpha)$ can be refined by a mathematical integer n to form the type n @ $\mathsf{int}(\alpha)$ that represents the singleton set $\{n\}$ of α-sized integers. The type ϕ @ bool is the single Boolean value reflecting the validity of proposition ϕ. The refinement type ℓ @ $\&_{\mathsf{own}}(\tau)$ denotes an *owned* (non-aliased) pointer and its refinement ℓ specifies

$$n_{\text{size}} \ @\ \text{alloc}_{\text{data}} \triangleq \text{struct}\ [n_{\text{size}} \ @\ \text{int}(\text{size\_t}),\ \&_{\text{own}}(\text{uninit}(n_{\text{size}}))]$$
$$\text{alloc}_{\text{spec}} \triangleq \text{fn}(\forall(n_{\text{len}}, n_{\text{size}}, \text{p}).\ \text{p} \ @\ \&_{\text{own}}(n_{\text{len}} \ @\ \text{alloc}_{\text{data}}),\ n_{\text{size}} \ @\ \text{int}(\text{size\_t}); \text{True})$$
$$\to \exists().\ (n_{\text{size}} \le n_{\text{len}}) \ @\ \text{optional}(\&_{\text{own}}(\text{uninit}(n_{\text{size}})),\ \text{null});$$
$$\text{p} \lhd_l ((n_{\text{size}} \le n_{\text{len}})\ ?\ (n_{\text{len}} - n_{\text{size}}) : n_{\text{len}}) \ @\ \text{alloc}_{\text{data}}$$

Figure 4. The formal specification of `alloc` (Figure 1) in RefinedC's type system (slightly simplified).

Type	Intuitive semantics
n @ int(α)	C integer of type α that encodes n
ϕ @ bool	C Boolean reflecting the truth of ϕ
ℓ @ $\&_{\text{own}}(\tau)$	unique ownership of τ at location ℓ
uninit(n)	n uninitialized (*i.e.*, arbitrary) bytes
ℓ @ ptr	pointer ℓ without ownership
null	singleton type of NULL
ϕ @ optional(τ_1, τ_2)	if ϕ then τ_1 else τ_2
wand(H, τ)	elements of τ with hole H
$\text{struct}_\sigma\ \overline{\tau}$	struct with layout σ, fields of types $\overline{\tau}$
$\exists x.\ \tau(x)$	type-level existential quantifier
$\{\tau \mid \phi\}$	elements of τ satisfying proposition ϕ
padded(τ, n)	elements of τ padded to n bytes

Table 1. A selection of RefinedC types.

the exact memory location that is owned. As examples, the annotations on `alloc_data` on line 3 in Figure 1 use $\&_{\text{own}}(\tau)$ together with uninit(n) to denote a pointer to a block of n bytes of uninitialized memory. In contrast, the type ℓ @ ptr represents a pointer *without* associated ownership of ℓ. The type ϕ @ optional(τ_1, τ_2) is a type-level case distinction on the validity of ϕ. It is most commonly used to represent nullable pointers (via $\&_{\text{own}}(\tau)$ and null), as illustrated in §2. Another interesting type is wand(H, τ), which is used to encode partial data structures via the magic wand [10]. We saw this type in the loop invariant of `free` in Figure 3.

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

Function types. Functions have RefinedC types of the form $\text{fn}(\forall x.\ \overline{\tau_{\text{arg}}}; H_{\text{pre}}) \to \exists y.\ \tau_{\text{ret}}; H_{\text{post}}$. Function types are generated from the source code annotations we have already seen. For example, the annotations on `alloc` (lines 6-9 of Figure 1) lead to the function type $\text{alloc}_{\text{spec}}$ shown in Figure 4. Logical variables in the `rc::parameters` annotation (line 6) correspond to x in the function type, the annotations `rc::args` and `rc::returns` (lines 7-8) correspond to $\overline{\tau_{\text{arg}}}$ and τ_{ret}, respectively, and the annotations `rc::requires` and `rc::ensures` (line 9) correspond to H_{pre} and H_{post}, respectively. Existential variables that are bound in the return type and the postconditions by `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 this fragment, called **Lithium**, describe how proof search works for it, and how we implement the proof search in Coq. We note that Lithium is similar to the substructural logic programming language Lolli [38] in its use of goal-directed search, but 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 Γ and Δ are two contexts of hypotheses whose elements can be used an arbitrary number of times (unrestricted) and at most once (resources), respectively. The syntax of contexts and goals is:

$$\begin{array}{lll}
\text{Atom} & A ::= \ell \lhd_l \tau \mid v \lhd_v \tau \mid \ldots & \\
\text{Basic goal} & F ::= \vdash^{\Sigma}_{\text{STMT}} s \mid A_1 <: A_2\ \{G\} \mid \ldots & \\
\text{Goal} & G ::= \text{True} \mid F \mid H * G \mid H \mathrel{-\!\!*} G \mid G_1 \wedge G_2 & \\
 & \quad \mid \forall x.\ G(x) \mid \exists x.\ G(x) & \\
\text{Left-goal} & H ::= \ulcorner \phi \urcorner \mid A \mid H * H \mid \exists x.\ H(x) & \\
\text{Contexts} & \Gamma ::= \emptyset \mid \Gamma, x \mid \Gamma, \phi \qquad \Delta ::= \emptyset \mid \Delta, A &
\end{array}$$

The unrestricted context Γ contains universally quantified variables (parameters) x and pure propositions ϕ, all of which are duplicable. The resource context Δ contains *atoms* A. The atom $\ell \lhd_l \tau$ expresses that location ℓ has type τ, and the atom $v \lhd_v \tau$ expresses that value v has type τ. Atoms are non-duplicable 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 $A_1 \mathrel{-\!\!*} (A_2 * G)$. The basic goal/typing judgment $\vdash^{\Sigma}_{\text{STMT}}\ s$ means that the C statement s is well-typed in the *function state* Σ, 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*, and Σ contains the postcondition of `alloc`, i.e., the consequent of $\text{alloc}_{\text{spec}}$.

$$\emptyset; \ell_{\text{d}} \triangleleft_l \text{p} \ @\ \&_{\text{own}}(n_{\text{len}} \ @\ \text{alloc}_{\text{data}}),\ \ell_{\text{size}} \triangleleft_l n_{\text{size}} \ @\ \text{int(size\_t)} \Vdash \left(\vdash^{\Sigma}_{\text{STMT}}\ \text{alloc}(\ell_{\text{d}}, \ell_{\text{size}})\right)$$

Besides basic goals, goals G may also contain the separation logic connectives $*$, $\mathrel{-\!\!*}$, $\wedge$, $\forall$ and $\exists$. However, the left sides of $\mathrel{-\!\!*}$ and $*$ are restricted to a smaller class of goals called *left goals*, H, which cannot contain $\wedge$, $\forall$ and $\mathrel{-\!\!*}$. 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. $G = G_1 \wedge G_2$: 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 $\frac{G'}{F'}$ whose conclusion F' can be unified with F, and prove $\Gamma; \Delta \Vdash G'$.

6. a. $G = (H_1 * H_2) * G'$: Prove the equivalent judgment $\Gamma; \Delta \Vdash H_1 * (H_2 * G')$; the next step will analyze the smaller formula H_1.
   b. $G = (\exists x.\, H(x)) * G'$: Prove the equivalent Lithium judgment $\Gamma; \Delta \Vdash \exists x.\, (H(x) * G')$ and use case (4); the next step will analyze a smaller formula $H(?x)$.
   c. $G = \ulcorner \phi \urcorner * G'$: Prove $\Gamma; \Delta \Vdash G'$ and emit a pure side condition ϕ to be solved under premises Γ.
   d. $G = A * G'$: Find $A' \in \Delta$ that is *related to* A, and prove $\Gamma; \Delta \setminus A' \Vdash A' <: A\ \{G'\}$. Atoms A and A' are related if they both assign types to the same value or location.
7. a. $G = (H_1 * H_2) \mathrel{-\!\!*} G'$: Prove the equivalent Lithium judgment $\Gamma; \Delta \Vdash H_1 \mathrel{-\!\!*} (H_2 \mathrel{-\!\!*} G')$; the next step will analyze the smaller formula H_1.
   b. $G = (\exists x.\, H(x)) \mathrel{-\!\!*} G'$: Prove the equivalent Lithium judgment $\Gamma; \Delta \Vdash \forall x.\, H(x) \mathrel{-\!\!*} G'$ and use case (3); the next step will analyze the smaller formula $H(y)$.
   c. $G = \ulcorner \phi \urcorner \mathrel{-\!\!*} G'$: Prove $\Gamma, \phi; \Delta \Vdash G'$.
   d. $G = A \mathrel{-\!\!*} G'$: Prove $\Gamma; \Delta, A \Vdash G'$.

No backtracking. The Lithium proof search procedure does not backtrack, which makes it efficient. Several design choices make this possible. First, the left side of $*$ in goals is limited to the form H, which cannot contain $\wedge$, $\forall$ and $\mathrel{-\!\!*}$. Without this restriction, proving a goal $G_1 * G_2$ would require a two-way split of the resource context Δ to prove G_1 and G_2 simultaneously, requiring backtracking over possible splits of Δ. 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 $\mathrel{-\!\!*}$ in goals is also restricted to the form H. This allows us to reduce local assumptions to atoms before adding them to the context Δ (case (7) and its subcases). By keeping only atoms in Δ, we eliminate backtracking over possible hypotheses that can be used to prove a given goal atom of the form $\ell \triangleleft_l \tau$ or $v \triangleleft_v \tau$: We trivially match ℓ or v from the goal to each hypothesis and at most one hypothesis will match, since we won't have multiple typing assumptions for the same location or value.

Finally, 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 match at most one typing rule.[5]

Implementation. We have implemented a Lithium interpreter in the Ltac language [22] of Coq. The interpreter maps Γ to the standard Coq context and Δ to the spatial context provided by the Iris Proof Mode [53, 55]. The search for matching RefinedC typing rules (case (5) above) is handled using Coq's typeclass mechanism [84].

Extensibility. Inspired by the *semantic typing* approach of RustBelt [41, 42], 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.

Judgment basics. RefinedC has a specialized typing judgment for each program construct, *e.g.*, $\vdash_{\text{IF}}$ for conditional statements and $\vdash_{\text{BINOP}}$ 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

[5] Lithium also offers a way to specify priority among RefinedC rules in case this property fails to hold.

$$\textsc{If-bool}\ \frac{(\ulcorner \phi \urcorner \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}} s_1) \land (\ulcorner \neg\phi \urcorner \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}} s_2)}{\vdash^{\Sigma}_{\text{IF}} \phi \ @\ \text{bool}\ \texttt{then}\ s_1\ \texttt{else}\ s_2} \qquad \textsc{If-int}\ \frac{(\ulcorner n \neq 0 \urcorner \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}} s_1) \land (\ulcorner n = 0 \urcorner \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}} s_2)}{\vdash^{\Sigma}_{\text{IF}} n \ @\ \text{int}(\alpha)\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

$$\textsc{T-if}\ \frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ \vdash^{\Sigma}_{\text{IF}} \tau\ \texttt{then}\ s_1\ \texttt{else}\ s_2\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2} \qquad \textsc{T-binop}\ \frac{\vdash_{\text{EXPR}} e_1\ \{v_1, \tau_1.\ \vdash_{\text{EXPR}} e_2\ \{v_2, \tau_2.\ \vdash_{\text{BINOP}} (v_1 : \tau_1) \odot (v_2 : \tau_2)\ \{v, \tau.\ G(v, \tau)\}\}\}}{\vdash_{\text{EXPR}} e_1 \odot e_2\ \{v, \tau.\ G(v, \tau)\}}$$

$$\textsc{O-optional-eq}\ \frac{(\ulcorner \phi \urcorner \mathrel{-\!\!*} v_1 \lhd_{\text{v}} \&_{\text{own}}(\tau) \mathrel{-\!\!*} G(\texttt{false}, \text{False @ bool})) \land (\ulcorner \neg\phi \urcorner \mathrel{-\!\!*} G(\texttt{true}, \text{True @ bool}))}{\vdash_{\text{BINOP}} (v_1 : \phi \ @\ \text{optional}(\&_{\text{own}}(\tau), \text{null})) = (v_2 : \text{null})\ \{v, \tau.\ G(v, \tau)\}}$$

$$\textsc{S-null}\ \frac{\ulcorner \neg\phi \urcorner * G}{v \lhd_{\text{v}} \text{null} <: v \lhd_{\text{v}} \phi \ @\ \text{optional}(\&_{\text{own}}(\tau), \text{null})\ \{G\}} \qquad \textsc{S-own}\ \frac{\ulcorner \phi \urcorner * (\forall \ell.\ \ell \lhd_l \tau_1 <: \ell \lhd_l \tau_2\ \{G\})}{v \lhd_{\text{v}} \&_{\text{own}}(\tau_1) <: v \lhd_{\text{v}} \phi \ @\ \text{optional}(\&_{\text{own}}(\tau_2), \text{null})\ \{G\}}$$

$$\textsc{O-add-uninit}\ \frac{\ulcorner 0 \le n_2 \le n_1 \urcorner * (v_1 \lhd_{\text{v}} \&_{\text{own}}(\text{uninit}(n_2)) \mathrel{-\!\!*} G(v_1 +_l n_2, \&_{\text{own}}(\text{uninit}(n_1 - n_2))))}{\vdash_{\text{BINOP}} (v_1 : \&_{\text{own}}(\text{uninit}(n_1))) + (v_2 : n_2 \ @\ \text{int}(\text{size\_t}))\ \{v, \tau.\ G(v, \tau)\}}$$

$$\textsc{Cas-bool}\ \frac{\begin{array}{c}(v_2 \lhd_{\text{v}} \&_{\text{own}}(\neg b_1 \ @\ \text{bool}) \mathrel{-\!\!*} G(\texttt{false}, \text{False @ bool})) \land \\ ((b_1\ ?\ H_\top : H_\bot) \mathrel{-\!\!*} (b_2\ ?\ H_\top : H_\bot) * (v_2 \lhd_{\text{v}} \&_{\text{own}}(b_1 \ @\ \text{bool}) \mathrel{-\!\!*} G(\texttt{true}, \text{True @ bool})))\end{array}}{\vdash_{\text{CAS}} \texttt{CAS}(v_1 : \text{atomicbool}(H_\top, H_\bot), v_2 : \&_{\text{own}}(b_1 \ @\ \text{bool}), v_3 : b_2 \ @\ \text{bool})\ \{v, \tau. G(v, \tau)\}}$$

Figure 5. Selected examples of RefinedC typing rules. (Simplified by *e.g.*, omitting refinements of $\&_{\text{own}}$.)

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 $\vdash_{\text{EXPR}} e\ \{v, \tau.\ G(v, \tau)\}$, 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 τ. 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 T-BINOP: Lithium first types e_1, then, in the continuation, types e_2, and, after both τ_1 and τ_2 have been inferred, introduces $\vdash_{\text{BINOP}}$. 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 NULL. Most uses of this pattern can be handled by three RefinedC typing rules: the rule O-OPTIONAL-EQ for comparing an optional with NULL, and the two rules S-NULL and S-OWN for introducing an optional type.

We first explain the rule O-OPTIONAL-EQ. This rule is used in proving $\vdash^{\Sigma}_{\text{STMT}}$ `if` $(e =$ `NULL`$)$ `then` s_1 `else` s_2. To do this, Lithium first applies T-IF, whose premise requires typing the boolean expression $e =$ `NULL`. It then applies T-BINOP, which requires typing e. Suppose Lithium infers the type ϕ @ optional$(\&_{\text{own}}(\tau), \text{null})$ for e. Next, Lithium types the second expression, `NULL`. This is trivial as `NULL` has type null. At this point, Lithium's goal is a judgment that matches the conclusion of O-OPTIONAL-EQ.

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

Returning to the example from above, the typing of $\vdash^{\Sigma}_{\text{STMT}}$ `if` $(e =$ `NULL`$)$ `then` s_1 `else` s_2 continues using IF-BOOL. This

[6] Sammler et al. [78, Section B] lists all RefinedC judgments and Sammler et al. [78, Section C] shows more typing rules for $\vdash^{\Sigma}_{\text{STMT}}$ and $\vdash_{\text{EXPR}}$.

rule also distinguishes two cases but one of these cases holds vacuously as one of ϕ and $\neg\phi$ is already in the context.

Next, we explain how Lithium establishes that a value v has type ϕ @ optional($\&_{\text{own}}(\tau)$, 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 types v. Typically, A' will type v at either null or $\&_{\text{own}}(\tau')$ for some τ'. In the first case, (6d) yields a new goal of the form $v \triangleleft_{\text{v}} \text{null} <: v \triangleleft_{\text{v}} (\phi$ @ optional($\&_{\text{own}}(\tau)$, null)) $\{G'\}$ (for some continuation G'). At this point, rule S-NULL is used to reduce the goal to proving $\neg\phi$ (and G'), which is what one expects from the intuitive meaning of the optional type. In the second case, Lithium's goal is $v \triangleleft_{\text{v}} \&_{\text{own}}(\tau') <: v \triangleleft_{\text{v}} (\phi$ @ optional($\&_{\text{own}}(\tau)$, null)) $\{G'\}$. Using rule S-OWN, this reduces to proving ϕ and a subsumption between τ' and τ, 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-UNINIT, which types the addition of an integer n_2 to a pointer to uninitialized memory of length n_1 (RefinedC type uninit(n_1)). The rule splits uninit(n_1) into the smaller pieces uninit(n_2) and uninit($n_2 - n_1$), after checking that $n_2 \leq n_1$. 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).

Fine-grained concurrency. RefinedC can also automatically verify fine-grained concurrent code. We illustrate this with the type atomicbool($H_\top, H_\bot$), which represents a Boolean that can be accessed atomically. The type holds the ownership of $H_\top$ if the Boolean is true, and $H_\bot$ 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 Caesium's CAS($\ell_{atom}, \ell_{exp}, v_{des}$) operation. The first argument (ℓ_{atom}) is a pointer to the value to be modified atomically, the second argument (ℓ_{exp}) is a pointer to the current expected value of ℓ_{atom}, and the third argument (v_{des}) 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 $H_\top$ or $H_\bot$. CAS-BOOL has two cases corresponding to whether the CAS fails and succeeds. (First case) When CAS fails, the second argument is updated to $\neg b_1$, 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 v_2, and the CAS returns `true`. (The implementation of the spinlock mentioned earlier uses CAS-BOOL with $b_1 \triangleq$ `false` and $b_2 \triangleq$ `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 the soundness of CAS-BOOL, which we have proved 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).

Table 2 shows how many typing rules Lithium applies automatically. This is relevant, seeing as typing rules perform tasks like ownership manipulation and unfolding of definitions that in some other tools are handled manually. The table 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++ [91] 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 be improved 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 that 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 then 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

Class	Test	Types	Rules	$\ulcorner\phi\urcorner$	Impl	Spec	Annot	Pure	Ovh.
#1	Singly linked list	wand, alloc	613	47/5	106	33	24	2	~0.2
	Queue	list segments, alloc	332	10/0	42	16	11	0	~0.3
	Binary search	arrays, func. ptr.	308	73/6	42	16	6	19	~0.6
#2	Thread-safe allocator	wand, padded, spinlock	319	28/2	68	18	21	3	~0.4
	Page allocator	padded	235	14/0	43	16	12	0	~0.3
#3	Binary search tree (layered)	wand, alloc	964	50/11	133	65	22	128	~1.1
	Binary search tree (direct)	wand, alloc	977	47/43	115	43	17	10	~0.2
#4	Linear probing hashmap	unions, arrays, alloc	1167	175/39	111	46	34	265	~2.7
#5	Hafnium's mpool allocator	wand, padded, spinlock	1730	122/11	191	56	52	5	~0.3
#6	Spinlock	atomic Boolean	65	14/1	24	12	13	1	~0.6
	One-time barrier	atomic Boolean	34	6/0	20	7	2	0	~0.1

Table 2. Evaluation of RefinedC. Types: Salient type constructs used. Rules: Number of typing rules applied by Lithium. $\ulcorner\phi\urcorner$: Number of side conditions automatically solved / manually solved. Impl: Lines of C code (counted by tokei [93]). Spec: Lines of top-level (function) specification. Annot: Lines of annotations in source code (*e.g.*, loop and data-structure invariants). Pure: Lines of pure Coq reasoning including definitions and lemma statements. Ovh.: Sum of Annot and Pure divided by Impl.

of partial data structures with magic wand [10, 11] 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 annotation overhead for both examples is low (mostly loop invariants). 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.

#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). (A third, real-world memory allocator 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 [33, 62]. 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 smaller 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 allocator[7] of the Hafnium hypervisor [34]. 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 spatial 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

[7] The original code had to be adapted since it uses integer-pointer casts, which are not yet supported by Caesium.

spinlock and the barrier. The initialization function needs manual proofs where it allocates a ghost token and for instantiating one existential quantifier. Decoupling the spinlock from the resources protected by it via the `spinlocked` type mentioned in §2 additionally requires 162 lines of Iris proofs. All together, this results in a reusable spinlock abstraction, which is used by other examples in Table 2 (the first allocator of #2, and the allocator of #5).

8 Related Work

Bedrock. Like RefinedC, the Bedrock project [12–14, 63] targets foundational and mostly automatic separation-logic based verification of low-level programs. However, Bedrock is based around a custom assembly-like language and custom DSLs built on top of it using macros that are verified similar to compiler passes [13, 14]. In contrast, RefinedC applies to existing C code that can be compiled using off-the-shelf optimizing C compilers. Bedrock specifications and abstract predicates are written in plain separation logic instead of a higher abstraction like RefinedC's type system. Bedrock's proof automation [12, 63] can be extended through hints for unfolding abstract predicates and with custom Ltac tactics. However, Bedrock's hint format is less expressive than Lithium, *e.g.*, it cannot represent rules like O-ADD-UNINIT or 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 [90], while no such hints are required in RefinedC. (Both tools require loop invariant annotations.)

VST. VST [2, 9] is a separation logic-based framework for verifying CompCert C programs. Users of VST deploy a set of semiautomatic tactics to build functional correctness proofs in Coq [9], or a front end [101] 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 insantiating existential quantifiers—things that RefinedC's Lithium-based automation handles automatically in most cases. As a concrete example, a verification of a binary search tree similar to the one in §7 by the authors of VST [96] 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 [50] and CertiKOS [31–33]. seL4 [49, 50] demonstrated the first formal proof of functional correctness of a complete, general-purpose operating-system kernel and comes with a translation-validation procedure [67, 81] to transfer the proofs to generated assembly code. However, most proofs about its C code are manual and rely only on basic tactic support [49, 103]. Later work automates some but not all of the most tedious parts [29, 30]. 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 [31–33] 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 CertiKOS 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 require such manual guidance in Coq in most cases, and it also supports a (different) version of the layer-based approach. However, further work is clearly 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 [16] 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 [17, 94]. No such ghost fields and annotations are necessary in RefinedC.

VeriFast [39] is an automated, separation logic-based verification tool for C and Java. It provides heuristics to automatically infer annotations to lower the proof overhead [99]. VeriFast's symbolic execution approach (of which only a core subset has been proven sound [98]) 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++[91]: 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 [95] by judicious use of existing lemmas and tactics. Other than this, the annotation burden is similar.

MatchC [77, 85] is an automated verification tool for C based on the K framework [76] and matching logic [75]. Its rewrite-based approach provides good automation for non-trivial pointer-manipulating programs and can be extended with new abstractions with custom rules similar to RefinedC. However, unlike RefinedC, these abstractions and their rules are not proven sound against a model, and thus 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. 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. Fiat Crypto [25] 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* [71] provides a semi-foundational approach to C verification through a shallow embedding of C in F* [87], 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 [106].

Separation logic automation. The literature abounds in (non-foundational) automatic solvers for separation logic and frame inference [15, 57, 59, 69, 72, 89]. 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 [15]) 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 by idiomatic C code.

Logic programming languages for linear and separation logic. Prior work on logic programming for linear or separation logic [1, 3, 35, 38] 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, 8, 105] focuses instead on the different—and simpler—problem of automatically verifying memory safety. One popular approach [19, 24, 68] 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 [74] verify memory safety of C code using a combination of refinement types [73] and alias types [83, 100]. 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. [74, Section 5.1]). Finally, safety can also be attained by using a memory-safe language such as Vault [23], Cyclone [40, 88], or Rust [92] in place of C. However, these languages rely on runtime checks, and, unlike RefinedC, their type systems only guarantee safety, not functional correctness.

Refinement and ownership type systems. Refinement types [27, 73, 104], although originally developed for functional programs, have also been used for the safety and correctness of imperative code [4, 74, 97]. 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 (*e.g.*, including pointer arithmetic, uninitialized memory, and concurrency). A recent, closely related piece of work in this area is ConSORT [97], which, like RefinedC, combines refinement types with ownership types. ConSORT achieves a higher degree of automation by using a simpler model of ownership types. In turn, ConSORT does not support abstractions like the magic wand and atomic Booleans that are used by 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 [64], and Iris [45, 55]. 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 other 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 [41]. However, the concrete design of RefinedC's type system differs in key respects: (1) RefinedC uses Mezzo-like [70] alias types [83, 100] 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.

Acknowledgments

We wish to thank Ralf Jung and Jan-Oliver Kaiser for their feedback and helpful discussions. 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 (grant agreement No 789108, ELVER), 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 a generous gift from Google. Robbert Krebbers was supported by the Dutch Research Council (NWO), project 016.Veni.192.259.

References

[1] Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *J. Log. Comput.* 2, 3 (1992), 297–347. https://doi.org/10.1093/logcom/2.3.297

[2] Andrew W. Appel. 2014. *Program Logics - for Certified Compilers*. Cambridge University Press. https://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers

[3] Pablo A. Armelín and David J. Pym. 2001. Bunched Logic Programming. In *IJCAR (LNCS, Vol. 2083)*. Springer, 289–304. https://doi.org/10.1007/3-540-45744-5_21

[4] Alexander Bakst and Ranjit Jhala. 2016. Predicate Abstraction for Linked Data Structures. In *VMCAI (LNCS, Vol. 9583)*. Springer, 65–84. https://doi.org/10.1007/978-3-662-49122-5_3

[5] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. ACM, 55–66. https://doi.org/10.1145/1926385.1926394

[6] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2004. A Decidable Fragment of Separation Logic. In *FSTTCS (LNCS, Vol. 3328)*. Springer, 97–109. https://doi.org/10.1007/978-3-540-30538-5_9

[7] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO (LNCS, Vol. 4111)*. Springer, 115–137. https://doi.org/10.1007/11804192_6

[8] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *POPL*. ACM, 289–300. https://doi.org/10.1145/1480881.1480917

[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 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

[10] Qinxiang Cao, Shengyi Wang, Aquinas Hobor, and Andrew W. Appel. 2019. Proof Pearl: Magic Wand as Frame. *CoRR* abs/1909.08789 (2019). http://arxiv.org/abs/1909.08789

[11] Arthur Charguéraud. 2016. Higher-order representation predicates in separation logic. In *CPP*. ACM, 3–14. https://doi.org/10.1145/2854065.2854068

[12] Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*. ACM, 234–245. https://doi.org/10.1145/1993498.1993526

[13] Adam Chlipala. 2013. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In *ICFP*. ACM, 391–402. https://doi.org/10.1145/2500365.2500592

[14] Adam Chlipala. 2015. From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification. In *POPL*. ACM, 609–622. https://doi.org/10.1145/2676726.2677003

[15] Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh. 2015. Automatic induction proofs of data-structures in imperative programs. In *PLDI*. ACM, 457–466. https://doi.org/10.1145/2737924.2737984

[16] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *TPHOLs (LNCS, Vol. 5674)*. Springer, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

[17] Ernie Cohen, Mark A. Hillebrand, Stephan Tobies, Michał Moskal, and Wolfram Schulte. 2012. Verifying C Programs: A VCC Tutorial. https://archive.codeplex.com/projects/VCC/fda99f81-18b5-45ae-8f49-5b28c747dcc3

[18] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. 2009. Unifying type checking and property checking for low-level code. In *POPL*. ACM, 302–314. https://doi.org/10.1145/1480881.1480921

[19] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. 2007. Dependent Types for Low-Level Programming. In *ESOP (LNCS, Vol. 4421)*. Springer, 520–535. https://doi.org/10.1007/978-3-540-71316-6_35

[20] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C - A Software Analysis Perspective. In *SEFM (LNCS, Vol. 7504)*. Springer, 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

[21] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

[22] David Delahaye. 2000. A Tactic Language for the System Coq. In *LPAR (LNCS, Vol. 1955)*. Springer, 85–95. https://doi.org/10.1007/3-540-44404-1_7

[23] Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-Level Protocols in Low-Level Software. In *PLDI*. ACM, 59–69. https://doi.org/10.1145/378795.378811

[24] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *SecDev*. IEEE Computer Society, 53–60. https://doi.org/10.1109/SecDev.2018.00015

[25] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *IEEE Symposium on Security and Privacy*. IEEE, 1202–1219. https://doi.org/10.1109/SP.2019.00005

[26] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. 2008. Combining Domain-Specific and Foundational Logics to Verify Complete Software Systems. In *VSTTE (LNCS, Vol. 5295)*. Springer, 54–69. https://doi.org/10.1007/978-3-540-87873-5_8

[27] Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *PLDI*. ACM, 268–277. https://doi.org/10.1145/113445.113468

[28] Dan Frumin, Léon Gondelman, and Robbert Krebbers. 2019. Semi-automated Reasoning About Non-determinism in C Expressions. In *ESOP (LNCS, Vol. 11423)*. Springer, 60–87. https://doi.org/10.1007/978-3-030-17184-1_3

[29] David Greenaway, June Andronick, and Gerwin Klein. 2012. Bridging the Gap: Automatic Verified Abstraction of C. In *ITP (LNCS, Vol. 7406)*. Springer, 99–115. https://doi.org/10.1007/978-3-642-32347-8_8

[30] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. 2014. Don't sweat the small stuff: formal verification of C code without the pain. In *PLDI*. ACM, 429–439. https://doi.org/10.1145/2594291.2594296

[31] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *POPL*. ACM, 595–608. https://doi.org/10.1145/2676726.2676975

[32] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. 2019. Building certified concurrent OS kernels. *Commun. ACM* 62, 10 (2019), 89–99. https://doi.org/10.1145/3356903

[33] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *PLDI*. ACM, 646–661. https://doi.org/10.1145/3192366.3192381

[34] Hafnium. 2020. Hafnium. https://review.trustedfirmware.org/plugins/gitiles/hafnium/hafnium/+/HEAD/README.md.

[35] James Harland, David J. Pym, and Michael Winikoff. 1996. Programming in Lygon: An Overview. In *AMAST (LNCS, Vol. 1101)*. Springer, 391–405. https://doi.org/10.1007/BFb0014329

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

[37] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP (LNCS, Vol. 4960)*. Springer, 353–367. https://doi.org/10.1007/978-3-540-78739-6_27

[38] Joshua S. Hodas and Dale Miller. 1991. Logic Programming in a Fragment of Intuitionistic Linear Logic. In *LICS*. IEEE Computer Society, 32–42. https://doi.org/10.1109/LICS.1991.151628

[39] 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 (LNCS, Vol. 6617)*. Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

[40] Trevor Jim, Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX*. 275–288. http://www.usenix.org/publications/library/proceedings/usenix02/jim.html

[41] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

[42] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2020. Safe systems programming in Rust: The promise and the challenge. *To appear in CACM* (2020). https://iris-project.org/pdfs/2020-rustbelt-cacm-submission.pdf

[43] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. ACM, 256–269. https://doi.org/10.1145/2951913.2951943

[44] 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

[45] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

[46] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[47] 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 (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

[48] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A formal C memory model supporting integer-pointer casts. In *PLDI*. ACM, 326–335. https://doi.org/10.1145/2737924.2738005

[49] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.* 32, 1 (2014), 2:1–2:70. https://doi.org/10.1145/2560537

[50] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *SOSP*. ACM, 207–220. https://doi.org/10.1145/1629575.1629596

[51] Robbert Krebbers. 2014. An operational and axiomatic semantics for non-determinism and sequence points in C. In *POPL*. ACM, 101–112. https://doi.org/10.1145/2535838.2535878

[52] Robbert Krebbers. 2015. *The C standard formalized in Coq.* Ph.D. Dissertation. Radboud University Nijmegen. https://robbertkrebbers.nl/thesis.html

[53] 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 (2018), 77:1–77:30. https://doi.org/10.1145/3236772

[54] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS, Vol. 10201)*. Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[55] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. ACM, 205–217. http://dl.acm.org/citation.cfm?id=3009855

[56] Robbert Krebbers and Freek Wiedijk. 2013. Separation Logic for Non-local Control Flow and Block Scope Variables. In *FoSSaCS (LNCS, Vol. 7794)*. Springer, 257–272. https://doi.org/10.1007/978-3-642-37075-5_17

[57] Quang Loc Le, Jun Sun, and Shengchao Qin. 2018. Frame Inference for Inductive Entailment Proofs in Separation Logic. In *TACAS (1) (LNCS, Vol. 10805)*. Springer, 41–60. https://doi.org/10.1007/978-3-319-89960-2_3

[58] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming undefined behavior in LLVM. In *PLDI*. ACM, 633–647. https://doi.org/10.1145/3062341.3062343

[59] Wonyeol Lee and Sungwoo Park. 2014. A proof system for separation logic with magic wand. In *POPL*. ACM, 477–490. https://doi.org/10.1145/2535838.2535871

[60] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert memory model, version 2.* Technical Report RR-7987. Inria. https://hal.inria.fr/hal-00703441

[61] Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J. Autom. Reason.* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

[62] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. 2020. Armada: low-effort verification of high-performance concurrent programs. In *PLDI*. ACM, 197–210. https://doi.org/10.1145/3385412.3385971

[63] Gregory Malecha, Adam Chlipala, and Thomas Braibant. 2014. Compositional Computational Reflection. In *ITP (LNCS, Vol. 8558)*. Springer, 374–389. https://doi.org/10.1007/978-3-319-08970-6_24

[64] William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A verified messaging system. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 87:1–87:28. https://doi.org/10.1145/3133911

[65] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.* 3, POPL (2019), 67:1–67:32. https://doi.org/10.1145/3290380

[66] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. In *PLDI*. ACM, 1–15. https://doi.org/10.1145/2908080.2908081

[67] Magnus Oskar Myreen. 2009. *Formal verification of machine-code programs*. Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.611450

[68] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: type-safe retrofitting of legacy code. In *POPL*. ACM, 128–139. https://doi.org/10.1145/503272.503286

[69] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. Automating Separation Logic with Trees and Data. In *CAV (LNCS, Vol. 8559)*. Springer, 711–728. https://doi.org/10.1007/978-3-319-08867-9_47

[70] François Pottier and Jonathan Protzenko. 2013. Programming with permissions in Mezzo. In *ICFP*. ACM, 173–184. https://doi.org/10.1145/2500365.2500598

[71] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F. *Proc. ACM Program. Lang.* 1, ICFP (2017), 17:1–17:29. https://doi.org/10.1145/3110261

[72] Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. 2016. A Decision Procedure for Separation Logic in SMT. In *ATVA (LNCS, Vol. 9938)*. 244–261. https://doi.org/10.1007/978-3-319-46520-3_16

[73] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *PLDI*. ACM, 159–169. https://doi.org/10.1145/1375581.1375602

[74] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. In *POPL*. ACM, 131–144. https://doi.org/10.1145/1706299.1706316

[75] Grigore Rosu, Chucky Ellison, and Wolfram Schulte. 2010. Matching Logic: An Alternative to Hoare/Floyd Logic. In *AMAST (LNCS, Vol. 6486)*. Springer, 142–162. https://doi.org/10.1007/978-3-642-17796-5_9

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

[77] Grigore Rosu and Andrei Stefanescu. 2012. Checking reachability using matching logic. In *OOPSLA*. ACM, 555–574. https://doi.org/10.1145/2384616.2384656

[78] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2020. RefinedC: An Extensible Refinement Type System for C Based on Separation Logic Programming (Appendix). https://plv.mpi-sws.org/refinedc/appendix.pdf

[79] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2020. RefinedC: An Extensible Refinement Type System for C Based on Separation Logic Programming (Artifact). https://plv.mpi-sws.org/refinedc/the_artifact.tgz Git repository: https://gitlab.mpi-sws.org/iris/refinedc/-/tree/f5211f79098e7c72dcc525a82a7c91a3e902f6b4.

[80] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *PLDI*. ACM, 77–87. https://doi.org/10.1145/2737924.2737964

[81] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *PLDI*. ACM, 471–482. https://doi.org/10.1145/2491956.2462183

[82] Zhong Shao, Valery Trifonov, Bratin Saha, and Nikolaos Papaspyrou. 2005. A type system for certified binaries. *ACM Trans. Program. Lang. Syst.* 27, 1 (2005), 1–45. https://doi.org/10.1145/1053468.1053469

[83] Frederick Smith, David Walker, and J. Gregory Morrisett. 2000. Alias Types. In *ESOP (LNCS, Vol. 1782)*. Springer, 366–381. https://doi.org/10.1007/3-540-46425-5_24

[84] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *TPHOLs (LNCS, Vol. 5170)*. Springer, 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

[85] Andrei Stefanescu. 2014. MatchC: A Matching Logic Reachability Verifier Using the K Framework. *Electron. Notes Theor. Comput. Sci.* 304 (2014), 183–198. https://doi.org/10.1016/j.entcs.2014.05.010

[86] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP (LNCS, Vol. 8410)*. Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

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

[88] Nikhil Swamy, Michael W. Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe manual memory management in Cyclone. *Sci. Comput. Program.* 62, 2 (2006), 122–144. https://doi.org/10.1016/j.scico.2006.02.003

[89] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2018. Automated lemma synthesis in symbolic-heap separation logic. *Proc. ACM Program. Lang.* 2, POPL (2018), 9:1–9:29. https://doi.org/10.1145/3158097

[90] The Bedrock Team. 2015. Verification of a singly linked list. https://github.com/mit-plv/bedrock/blob/e3ff3c2cba9976ac4351caaabb4bf7278bb0dcbd/Bedrock/Examples/SinglyLinkedList.v.

[91] The Coq-std++ Team. 2020. An extended "standard library" for Coq. https://gitlab.mpi-sws.org/iris/stdpp

[92] The Rust Team. 2020. The Rust programming language. https://rust-lang.org.

[93] The Tokei Team. 2020. Tokei. https://github.com/XAMPPRocky/tokei.

[94] The VCC Team. 2016. Verification of a singly linked list. https://github.com/microsoft/vcc/blob/47f3f33d459f5fd9233203ec3d5d2fc8032b7db5/vcc/Docs/Tutorial/c/7.2.list.c.

[95] The Verifast Team. 2019. Verification of a binary search tree. https://github.com/verifast/verifast/blob/8bc966726de829749eaf916ec3863bf2947dcc37/examples/sorted_bintree.c.

[96] The VST Team. 2020. Verification of Binary Search Tree. https://github.com/PrincetonUniversity/VST/blob/14e6b3a79a9685a478786436c6f0a45dc44c3d52/progs/verif_bst.v.

[97] John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. 2020. ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs. In *ESOP (LNCS, Vol. 12075)*. Springer, 684–714. https://doi.org/10.1007/978-3-030-44914-8_25

[98] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2015. Featherweight VeriFast. *Log. Methods Comput. Sci.* 11, 3 (2015). https://doi.org/10.2168/LMCS-11(3:19)2015

[99] Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans. 2011. Annotation Inference for Separation Logic Based Verifiers. In *FMOODS/FORTE (LNCS, Vol. 6722)*. Springer, 319–333. https://doi.org/10.1007/978-3-642-21461-5_21

[100] David Walker and J. Gregory Morrisett. 2000. Alias Types for Recursive Data Structures. In *TIC (LNCS, Vol. 2071)*. Springer, 177–206. https://doi.org/10.1007/3-540-45332-6_7

[101] Qinshi Wang and Qinxiang Cao. 2019. VST-A: A Foundationally Sound Annotation Verifier. *CoRR* abs/1909.00097 (2019). http://arxiv.org/abs/1909.00097

[102] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined behavior: what happened to my code?. In *APSys*. ACM, 9. https://doi.org/10.1145/2349896.2349905

[103] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. 2009. Mind the Gap. In *TPHOLs (LNCS, Vol. 5674)*. Springer, 500–515. https://doi.org/10.1007/978-3-642-03359-9_34

[104] Hongwei Xi. 2007. Dependent ML An approach to practical programming with dependent types. *J. Funct. Program.* 17, 2 (2007), 215–286. https://doi.org/10.1017/S0956796806006216

[105] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. 2008. Scalable Shape Analysis for Systems Code. In *CAV (LNCS, Vol. 5123)*. Springer, 385–398. https://doi.org/10.1007/978-3-540-70545-1_36

[106] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *CCS*. ACM, 1789–1806. https://doi.org/10.1145/3133956.3134043