RefinedC: An Extensible Refinement Type System for C Based on Separation Logic Programming

MICHAEL SAMMLER, MPI-SWS, Germany
RODOLPHE LEPIGRE, MPI-SWS, Germany
ROBBERT KREBBERS, Delft University of Technology, The Netherlands
KAYVAN MEMARIAN, University of Cambridge, UK
DEREK DREYER, MPI-SWS, Germany
DEEPAK GARG, MPI-SWS, Germany

Although the C and C++ programming languages are by their very nature unsafe, they nevertheless power most of the lower layers of the software stack, including hypervisors and operating systems. As a consequence, many critical systems remain vulnerable to safety and functional correctness issues, which can lead to crashes and security breaches. To improve on this situation, we propose a new approach to verifying the correctness of idiomatic C code, called **RefinedC**—a type system combining ownership types (to ensure memory safety) with refinement types (to ensure functional correctness). RefinedC is built atop a new "separation logic programming" language we call Lithium, which supports automatic proof search by backchaining, but without the need for backtracking. By virtue of its representation as a Lithium program, RefinedC's type system is fundamentally *extensible*, meaning that the set of typing rules is not fixed and can be grown over time by adding new clauses to the Lithium program. We show that this extensibility is key to supporting numerous low-level idioms (*e.g.*, involving pointer manipulations) which C programmers employ in practice. RefinedC and Lithium are embedded in the Iris framework for concurrent separation logic in Coq, allowing us to foundationally certify the result of the RefinedC type checker, including user-defined extensions.

1 INTRODUCTION

Despite numerous advances in programming language technology over the past several decades, much systems software, including safety- and security-critical software, is still programmed in the low-level languages C and C++. These languages remain widely used in large part because they provide 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 to make their C/C++ code functionally correct and—ideally—to *prove* that this is the case [Austin et al. 1994; Necula et al. 2002; Condit et al. 2007, 2008; Cohen et al. 2009; Rondon et al. 2010; Chlipala 2011, 2015; Jacobs et al. 2011; Greenaway et al. 2013; Krebbers 2015; Kirchner et al. 2015; Elliott et al. 2018; Cao et al. 2018; Frumin et al. 2019; Lorch et al. 2020].

In this paper, we explore a novel approach to the problem of verifying correctness of C code, which we call **RefinedC**. At a high level, RefinedC is motivated by the following goals:

Verifying idiomatic C code. A number of prior approaches to provably improving the reliability of C code have adopted the strategy of changing the semantics of the language, either by judiciously inserting run-time checks into C code (usually to ensure memory safety [Austin et al. 1994; Condit et al. 2007; Elliott et al. 2018]) or modifying the representation of C data types [Necula et al. 2002]. Other approaches leave the language semantics alone but avoid dealing with some important

Draft of July 2020. Authors' addresses: Michael Sammler, MPI-SWS, Saarland Informatics Campus, Germany, msammler@mpi-sws.org; Rodolphe Lepigre, MPI-SWS, Saarland Informatics Campus, Germany, lepigre@mpi-sws.org; Robbert Krebbers, Delft University of Technology, The Netherlands, mail@robbertkrebbers.nl; Kayvan Memarian, University of Cambridge, UK, kayvan.memarian@cl.cam.ac.uk; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi- sws.org; Deepak Garg, MPI-SWS, Saarland Informatics Campus, Germany, dg@mpi-sws.org.

features of C like arithmetic overflow, the & (address-of) operator, uninitialized memory, and concurrency with data races [Condit et al. 2008; Rondon et al. 2010]. Seeing as C/C++ (in all its low-level glory) is likely to remain the *lingua franca* of systems programming for the foreseeable future, RefinedC focuses on verifying a range of realistic programming idioms in C code directly, with minimal changes to C's semantics, keeping it as efficient as it actually is.

Extensible automation. In order to scale to support the verification of large code bases, it is essential to offer a high degree of proof automation. However, since C gives the programmer so much freedom in how pointers are manipulated, it seems nigh impossible to build a fixed system that automatically accounts for all the low-level idioms that C programmers employ in practice.

RefinedC tackles this challenge using an *extensible, ownership-based, refinement type system*:

- *Types* are arguably the most scalable and widely deployed of all automated formal methods, and the most familiar to programmers. We use types to carefully constrain and predictably guide the proof search, as well as to provide useful error messages in case of a failed proof.
- *Ownership-based types* (following the spirit, though not the detail, of Rust) make it possible to reason modularly about mutation of heap-allocated data structures, even in the presence of aliasing and concurrency.
- *Refinement types* make it possible to verify full functional correctness of C programs by annotating data types with essential functional properties of the data they classify.
- The type system of RefinedC is not fixed but rather *extensible*: the user can extend the system with new (refinement) types and corresponding typing rules over time, so that they can custom-tailor the proof search to handle new programming idioms as needed.

As a result of these design choices, verification using RefinedC is *highly automatic*: the RefinedC type checker can be taught how to automatically discharge complex pointer and ownership reasoning using only standard annotations (*i.e.*, invariants of data structures, function specifications, and loop invariants), even for non-standard low-level programming patterns.

Foundational certification in Coq. Several verification systems for C are built as standalone tools, which offers them flexibility in terms of automation. However, as the logical foundations of these tools grow more complex, and as their bespoke proof automation becomes more sophisticated, their trusted computing base (TCB) grows larger too. This is of particular concern given the emphasis we have placed on extensibility: if we want users to be able to extend RefinedC's automation to support new programming idioms, how do we ensure that their extensions are sound?

To address this concern, RefinedC type checking is *foundationally certified* [Anonymous 2020], meaning that it outputs an independently-checkable proof of correctness in the Coq proof assistant [The Coq Team 2020]. In particular, following the approach of RustBelt [Jung et al. 2018a], RefinedC adopts a *semantic* approach to typing: rather than defining typing syntactically via a fixed set of inference rules, we interpret RefinedC types as predicates in Iris [Jung et al. 2015, 2016; Krebbers et al. 2017a; Jung et al. 2018b], a modern dialect of separation logic [O'Hearn et al. 2001; Reynolds 2002] implemented in Coq [Krebbers et al. 2017b, 2018]. As in the case of RustBelt, Iris is a good fit for modeling RefinedC's ownership-based types since, being a separation logic, it provides built-in support for reasoning about ownership. The soundness of RefinedC typing rules—including those defined by the user—can then be established by proving lemmas about the semantic model of types in Iris. In the end, if RefinedC type checking of a program P succeeds, we obtain an Iris proof that P satisfies the semantic predicate interpretation of its RefinedC type, which by Iris's soundness theorem in Coq implies that P is safe to execute and adheres to its specification. Thus, there is no need to trust that RefinedC's proof automation has been implemented correctly—one need only trust the program's specification, our C semantics, and Coq.

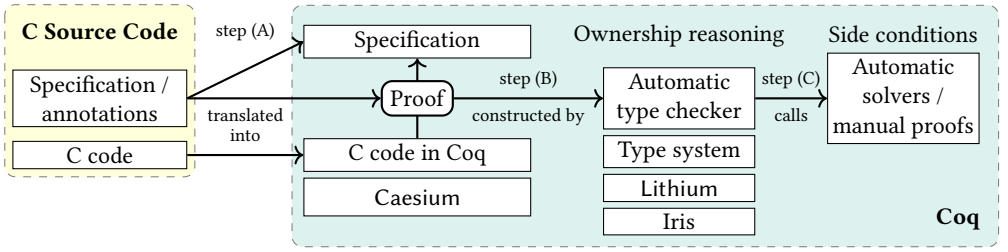

Fig. 1. The architecture of RefinedC.

1.1 Key Idea: Encoding the RefinedC Type System as a Separation Logic Program

The key idea behind RefinedC that enables automation is that its type system is encoded in a *logic programming framework* for the separation logic of Iris that we call **Lithium**.

Lithium is a carefully restricted subset of Iris propositions, on which efficient *goal-directed* proof search (also called *backchaining* search) is possible. We have implemented an interpreter for Lithium using Coq tactics. 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 automatic and foundationally-sound method for checking C programs against RefinedC types, and one which is inherently extensible by virtue of its representation as an open set of Lithium rules.

Lithium's interpreter is inspired by prior work in logic programming, notably Lolli [Hodas and Miller 1994]. However, Lithium differs from this prior work in that the set of propositions admitted by Lithium is very carefully restricted to ensure that goal-directed proof search can be accomplished *without backtracking*. This restriction speeds up the search for RefinedC typing derivations and enables precise error localization when typing fails. The lack of backtracking does not reduce the expressiveness of RefinedC because the program's syntax already guides the proof search.

RefinedC *structurally.* Figure 1 is a structural diagram of the components of RefinedC. Developers write standard C code that they would write in the absence of 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 [Memarian et al. 2019]) translates the C code to a deep embedding of C in Coq, called **Caesium**, and 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 in Coq to produce a typing derivation proving the specification in Coq. During this process, verification conditions—which are 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 interactive proofs.

Under the hood, hidden from the ordinary C programmer, lie RefinedC's types and typing rules, which have been defined in Lithium by an expert ahead of time. The expert must define RefinedC types semantically (as explained above), and prove every rule sound in Iris against a Coq formalization of the C semantics, which we also provide.

RefinedC *in practice.* C programs can be classified into three categories based on difficulty of verification using RefinedC (in order of increasing difficulty):

```
struct [[rc::refined_by("nlen : nat")]] alloc_data {
  [[rc::field("nlen @ int<size_t>")]] size_t len;
  [[rc::field("&own<uninit<nlen>>")]] unsigned char* buffer;
};

[[rc::parameters("nlen : nat", "nsize : nat", "p : loc")]]
[[rc::args      ("p @ &own<nlen @ alloc_data>", "nsize @ int<size_t>")]]
[[rc::returns   ("{nsize ≤ nlen} @ optional<&own<uninit<nsize>>, null>")]]
[[rc::ensures   ("p @ &own<{if nsize ≤ nlen then nlen - nsize else nlen} @ alloc_data>")]]
void* alloc(struct alloc_data* d, size_t size) {
  if(size > d->len) return NULL;
  d->len -= size;
  return d->buffer + d->len;
}
```

Fig. 2. Allocator example (annotations slightly simplified).

(1) The programs that are easiest to verify are those that only use existing abstractions, *e.g.*, ownership manipulations, partial data structures, or concurrency abstractions. They can be verified automatically without programmer intervention, using a library of typing rules that we have already created. Verifying a program in this category only requires understanding of the RefinedC rules and annotations, and Coq's surface syntax for mathematical specifications, but no understanding of Coq proofs, Lithium, or Iris. As we show in §6, this category covers a large number of programs and programming patterns. In particular, it includes allocators (§2.1), linked list insertion (§2.3), and usage of a spinlock abstraction (§2.2).
(2) Programs whose specification requires defining new purely-functional structures in Coq or whose pure verification conditions cannot be discharged by RefinedC's default solver are slightly more challenging to verify. Indeed, they require knowledge of Coq but not Lithium or Iris. For example, programs requiring multiset reasoning (§2.3) fall in that category.
(3) Finally, the last category encompasses all remaining programs whose verification requires extending RefinedC's type system in Lithium. This requires familiarity with Lithium and Iris, but allows the definition of new abstractions (*e.g.*, concurrency abstractions like the spinlock used in §2.2) that can then be reused for verifications falling under earlier categories.

Contributions. In summary, this paper makes the following contributions:

- RefinedC (§4): A new, foundationally sound, automatic, and extensible approach to full functional verification of idiomatic C programs based on refinement and ownership types.
- Lithium (§5): A logic programming language based on Iris (separation logic), embedded and automated in Coq, suitable for encoding and extending the typing rules of RefinedC.
- Caesium (§3): A core low-level C language with a formal semantics, embedded in Coq, and equipped with a front end that elaborates C sources into Caesium definitions in Coq.
- A large library of typing rules for RefinedC, all verified in Iris against Caesium.
- The verification of several case studies of varying complexity (§6), to substantiate the claim that the RefinedC/Lithium approach scales to handle idiomatic C code.

2 REFINEDC BY EXAMPLE

In this section we use simple motivating examples to introduce RefinedC at an intuitive level, from the user's point of view. We first demonstrate RefinedC's annotation syntax by verifying a memory allocator (§2.1). This example involves the definition of a refinement type and a function specification. We then verify a derived thread-safe allocation function (§2.2) based on a predefined *spinlock* abstraction (also verified in RefinedC). This example makes use of the fact that the spinlock

interface allows the protected data to be decoupled from the lock itself. It also demonstrates the use of global variables. Finally, we verify the deallocation mechanism of a more complex memory allocator, which relies on a linked list of free chunks (§2.3). This example demonstrates the definition of a recursive refinement type (for the linked list of chunks), as well as loop invariant annotations.

2.1 A Simple Memory Allocator

Let us turn to the memory allocator defined in Figure 2, first ignoring RefinedC's annotations of the form `[[rc::...]]`.[1] The code starts with the definition of the type `struct alloc_data`, which represents the state of the allocator. It consists of a block of available memory, whose size is stored in the `len` field, and whose first byte is pointed to by the `buffer` field. The `alloc` function attempts to allocate `size` bytes of memory from a `struct alloc_data`. It uses the `len` field to check if enough memory is available, and returns `NULL` if that is not the case. If `buffer` is large enough, then its last `size` bytes are allocated using simple pointer arithmetic. Note that the `buffer` field does not need to be modified in the process: only its recorded length is updated.

It is intuitively clear that the `alloc` function behaves as an allocator, in the sense that it always returns a pointer to a different memory region (of the expected size), and it only fails when this is not possible (in which case the `struct alloc_data` is not modified).[2] We show in the following paragraphs how this informal correctness claim can be turned into a formal RefinedC specification that can be automatically verified by the RefinedC automation.

Allocator state invariant. The annotations on `struct alloc_data` define a new RefinedC type called `alloc_data`, which formalizes the invariant of the allocator state. Intuitively, what is important about the state of the allocator is how many bytes are available in its buffer. This is captured by the annotation `rc::refined_by` on line 1, which makes `alloc_data` parametric in a (mathematical) natural number `nlen` representing the number of available bytes.

The `rc::field` annotations make use of `nlen` when specifying the types of `alloc_data`'s fields. The fact that the `len` field gives the number of available bytes, and hence represents `nlen`, is encoded using the refinement type `nlen @ int<size_t>` in line 2. This type can be read as "`nlen` refines `int<size_t>`", and it is essentially a singleton type containing a `size_t` integer that encodes `nlen`. The `buffer` field is informally meant to point to a block of memory of size `nlen` whose content is meaningless. In RefinedC, this is formalized using the type `&own<uninit<nlen>>`, which represents an owned pointer to uninitialized (*i.e.*, arbitrary) memory of the specified size. An owned pointer is an exclusive resource, hence the pointed memory can only be accessed by a single thread at any given time, which implies that its owner does not have to worry about it being modified concurrently.

Let us now take a step back to understand the difference between the C type `struct alloc_data` and the RefinedC type `alloc_data`. On the one hand, `struct alloc_data` only specifies the *physical layout* of its inhabitants. In particular, it contains information about the names and the offsets of the fields, which is useful for the compiler to generate field accesses. However, this layout information does not give any meaningful correctness guarantees. For example, it does not enforce that the `len` field is a valid integer: it could very well be uninitialized. On the other hand, the RefinedC type `alloc_data` captures the invariant that is satisfied by the `struct alloc_data` values on which `alloc` operates. Note, however, that the additional specification information in the RefinedC type is purely logical: it does not influence how the code is executed.

To summarize, the annotations on the declaration of `struct alloc_data` in Figure 2 lead to the definition of a new RefinedC type named `alloc_data`, which is refined by the number of bytes

[1] RefinedC relies on C2x attributes for annotations, which are supported by all of the mainstream C compilers.
[2] To simplify the presentation we ignore memory alignment considerations in this section.

that are available for allocation. As a consequence, for any (mathematical) natural number `n`, the RefinedC type `n @ alloc_data` asserts exclusive ownership of `n` bytes of uninitialized memory.

***Specification of* `alloc`.** We now turn to the annotations on `alloc`, whose purpose is to assign a type (*i.e.*, a specification) to the function. Thanks to the `rc::parameters` annotation on line 6, our specification is parameterized by a number of logical variables. Parameters are universally quantified in the whole specification and, much like declarations of refinements on a `struct`, they can range over arbitrary mathematical domains (encoded as Coq types). The `alloc` function has three such parameters: the natural number `nlen` indicating the number of bytes available for allocation, another natural number `nsize` corresponding to the requested number of bytes, and the memory location `p` at which the allocation data is stored. An important role of parameters is to tie together the refinements contained in the argument and return types (as well as possible pre- and postconditions) of the function. We will see several examples of that in what follows.

The types of the two arguments of `alloc` are given using the `rc::args` annotation on line 7. The type `p @ &own<nlen @ alloc_data>` specifies that the first argument of the function is an owned pointer to an allocator state with `nlen` available bytes, and that is itself stored at location `p`. Then, similarly to the `len` field of `alloc_data`, the type `nsize @ int<size_t>` specifies that the second argument of `alloc` encodes the requested allocation size `nsize` as an integer.

Next, the return type of `alloc` is specified using `rc::returns` on line 8. Since the function may fail by returning `NULL`, it is not possible for its return type to be a bare owned pointer. Indeed, an owned pointer is always valid, while `NULL` clearly is not. To solve this problem, we rely on the `optional<...>` type, which represents either a (valid) owned pointer or `null` (a RefinedC singleton type representing `NULL`). The relevant case is determined by the refinement of the `optional<...>` type, which represents the condition under which the value is an owned pointer. In the return type of `alloc` the validity of condition `nsize ≤ nlen`[3] precisely determines if the requested allocation will succeed (*i.e.*, if the allocator state owns a large enough amount of memory).

The final part of the specification of `alloc` is the postcondition given by `rc::ensures` on line 9. Its role is to ensure that the function returns the ownership of the allocator state (that it received through its first argument) back to its caller. Note that ownership of the allocator state is returned with an updated refinement since the amount of available memory decrease if the allocation is successful (the update of the logical state is expressed using Coq syntax). On the other hand, it is important to ensure that the returned ownership concerns the *same* pointer as the one that was passed as an argument. This is why the first argument of the function, as well as its return type, are explicitly refined by the *same* location `p`. This pattern, *i.e.*, returning ownership via a postcondition and tying it to an argument via a refinement, is commonly used in RefinedC. It is inspired by Mezzo [Pottier and Protzenko 2013], and is an alternative to mutable references as used by Rust.

Verification. The annotations on `struct alloc_data` and `alloc` together precisely capture the intuitive correctness specification of the allocator, which can be *automatically verified* by RefinedC without further help. While many technical details make this verification challenging, the main difficulty is the splitting of the block of uninitialized memory (of the `buffer` field) into the newly allocated memory (that is returned to the caller) and the remaining region (that stays in the `buffer` field). Intuitively, this split happens during the pointer addition on line 13, after which the newly obtained pointer is associated with the ownership of the memory it points to. The extensible RefinedC type system captures this informal reasoning using a specifically tailored typing rule for `uninit<...>` (see O-ADD-UNINIT on page 17), and this is what allows verification to be fully automated.

[3]Inside RefinedC annotations curly braces { . . . } delimit quoted Coq code.

```
Cannot solve sidecondition in function "alloc" !
Location: "alloc.c" [ 13 : 2 - 13 : 28 ]
Case distinction (nsize > nlen) -> false at "alloc.c" [ 11 : 5 - 11 : 18 ]
...
H3 : ¬ nsize > nlen
----------------------------------------
nsize < nlen
```

Fig. 3. Error message for a buggy specification of `alloc` (slightly simplified).

```
1  [[rc::parameters("lid : lock_id")]]
2  [[rc::global("spinlock<lid>")]]
3  struct spinlock lock;
4
5  [[rc::parameters("lid : lock_id")]]
6  [[rc::global("spinlocked<lid, {\"data\"}, alloc_data>")]]
7  struct alloc_data data;
8
9  [[rc::parameters("lid : lock_id", "nsize : nat")]]
10 [[rc::args      ("nsize @ int<size_t>")]]
11 [[rc::requires  ("[initialized \"lock\" lid]", "[initialized \"data\" lid]")]]
12 [[rc::returns   ("optional<&own<uninit<nsize>>, null>")]]
13 void* thread_safe_alloc(size_t size) {
14   sl_lock(&lock);
15   rc_unwrap(data);
16   void* ret = alloc(&data, size);
17   sl_unlock(rc_wrap(&lock));
18   return ret;
19 }
```

Fig. 4. Thread-safe allocation function (annotations slightly simplified).

Error messages. Formal verification remains a challenging endeavor, and there are ample opportunities for user mistakes. As a consequence, it is crucial for a verification tool to behave predictably, and to give helpful error messages upon failure. And indeed, RefinedC's type system gives precise feedback. For example, let us see what happens if the user accidentally writes `nsize < nlen` instead of `nsize ≤ nlen` in the specification of `alloc` on line 8 in Figure 2. The problem here is that when `nsize = nlen` the implementation returns a valid pointer, while the specification expects `NULL`. As a consequence, verification fails, and RefinedC produces the error message displayed in Figure 3. It tells the programmer at what point in the code the verification failed (during the `return` statement on line 13), and in which branch of the `if` statement on line 11 (the else branch). Also, the error message shows the precise side condition that could not be proven. Here, `alloc` returns a valid pointer, so one must prove `nsize < nlen`, which is clearly impossible in this case. This precise error message gives the developer useful information for debugging this failed verification attempt.

2.2 Thread-Safety Using a Spinlock

According to the specification of `alloc` discussed in §2.1, a thread can only call the function if it has full ownership of the allocator state. And indeed, `alloc` is clearly subject to data races if used concurrently on the same `struct alloc_data`. To make the allocator thread safe, the obvious solution is to protect its global state using a lock, and that is exactly what has been done in the function `thread_safe_alloc` of Figure 4. The allocator state is stored in the global variable `data` (line 7), which is

protected by spinlock `lock` (line 3). The `thread_safe_alloc` function then simply acquires and releases the lock using `sl_lock` and `sl_unlock` around the call to `alloc` on `data`.[4]

Global variables. Before introducing the spinlock abstraction, we need to take a detour to explain the handling of global variables in RefinedC. Much like function arguments or `struct` fields, global variable must be annotated with a type. This type may (once again) depend on the logical variables specified with `rc::parameters`, and it is itself specified using `rc::global`. However, global variables are special in the sense that their specification (*i.e.*, their type) in only satisfied once they have been explicitly initialized (*e.g.*, by the `main` function).

As a consequence, when a function relies on some global variable being initialized, this fact must be made explicit in its specification with a precondition using the `rc::requires` annotation. Indeed, `thread_safe_alloc` has such a precondition for both global variables `lock` and `data` on line 11. Here, the separation logic assertions `initialized "lock" lid` and `initialized "data" lid`[5] specify that the variables have been initialized, and they also tie the `lid` parameter of the function to the parameter of the same name in the specification of both global variables. In particular, this enforces that the two global variables satisfy their specification for the *same* lock identifier.

Spinlock abstraction. The locking mechanism used in `thread_safe_alloc` is a simple spinlock that was previously verified in RefinedC, and that is used here as a library. The spinlock interface relies on two abstract types `spinlock<...>` and `spinlocked<...>`. The former is the type of a spinlock, and it is parameterized by a `lock_id`, *i.e.*, a unique identifier for a particular spinlock instance. The latter corresponds to the type of a value (whose type is given as third argument) that is protected by the lock identified by the first argument.[6] The main idea for using a lock is that the protected data can only be accessed (*i.e.*, the `spinlocked<...>` type stripped from their type) if a token associated to the lock has been obtained. This token is logically returned by `sl_lock` through a postcondition, and it must be given up when calling `sl_unlock` as it is required as a precondition.

Verification. Before we discuss the verification of `thread_safe_alloc`, it is worth pointing out that its specified return type differs from that of `alloc`. Indeed, due to concurrency, `thread_safe_alloc` cannot give any guarantees about whether it will succeed or not. As a consequence, the `rc::returns` annotation does not contain a refinement on the `optional<...>`.[7]

The main challenge for automating the verification of `thread_safe_alloc` using RefinedC has to do with the `spinlocked<...>` appearing in the type of the protected data. After the lock has been acquired, the `spinlocked<...>` type constructor must be stripped away before being able to use the protected data. Moreover, it must be reinstated before releasing the lock. Hence the question is: how can the type system decide when to unwrap, and then wrap again, the type of the protected data? This may seem like a simple question to answer, but there are several problems. For instance, there may be several different resources protected by the same lock, and not all of them may need to be unwrapped (or even be unwrappable). Also, the `spinlocked<...>` type may be hidden away behind abstractions. Hence, to keep the system as flexible as possible, it is the responsibility of the programmer to guide the type system using annotations. That is the reason for the use of the `rc_unwrap` and `rc_wrap` macros in the implementation of `thread_safe_alloc`.

[4]The `rc_unwrap` and `rc_wrap` macros expand to RefinedC annotations, and they are no-ops as far as C is concerned. Moreover, the `rc_wrap` macro is only explicitly included for clarity: it is automatically inserted by RefinedC.
[5]Inside RefinedC annotations square brackets `[...]` delimit quoted Iris propositions.
[6]The second argument of `spinlock<...>` is a string that uniquely identifies the object that is being protected. Indeed, with our spinlock abstraction *one* lock can protect, *e.g.*, multiple global variables.
[7]In fact it is implicitly refined by an existentially quantified proposition, which does not give any information.

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

Fig. 5. Allocator with freelist example (annotations slightly simplified).

2.3 Deallocation Using a List of Free Chunks

As our last example, we consider the implementation of the deallocation function `free` given in Figure 5. Its purpose is to insert the block of memory that is being freed into a linked list of memory chunks that are available for allocation. When in the list, every chunk is headed by a `struct chunk` giving its `size` on line 8, as well as a pointer to the `next` chunk on line 9, if there is one, or `NULL` otherwise. The remaining part of the block (after the header) is completely arbitrary.

An interesting invariant that is maintained by `free` is that the list of chunks it manipulates is sorted by ascending order of size. (This can be useful to reduce fragmentation.) As a consequence the function uses a loop on line 22-25 to find the right position where to insert the new chunk.

Recursive type definition. Similarly to the definition of `alloc_data` in §2.1, a type `chunks_t` is defined by annotating the declaration of `struct chunk`. However, the definition of `chunks_t` has a slightly more complex structure since `struct chunk` does not directly correspond to a list. Indeed, it only stores the contents of an internal node, and it cannot represent the empty list. The C type that really corresponds to a linked list is `struct chunk*`, which is given the name `chunks_t`. This indirection is introduced by the `rc::ptr_type` annotation on line 3, which specifies that the created `chunks_t` type is an optional pointer to the type specified by the remaining annotations on `struct chunk`.[8]

Another particularity of the `chunks_t` type is that its definition is recursive. Indeed, the type annotation on the `next` field mentions the type `chunks_t` itself. This is not really surprising because

[8] An ellipsis is used as a placeholder for the type specified by the remaining annotations.

lists are an example of recursive types. Importantly, the unfolding of recursive types is automatically handled by RefinedC: no particular annotations are required for that.

Multiset and invariant. Let us now have a closer look at the definition of the `chunks_t` type to understand the invariant that it specifies. The first thing to note is that the type is refined by a multiset of natural numbers `s` on line 2, which intuitively represents the size of all the chunks stored in the list. We use a multiset because there can be multiple chunks with the same size. Note that the size of chunks is the only useful information about them because their contents is arbitrary.

When a `chunks_t` does not represent the empty list, the **`struct`** that it points to is parameterized by the size of the current chunk `n` as well as the multiset `tail` refining the remainder of the list. These two parameters are given using the `rc::exists` notation, which suggests that they are existentially quantified. Indeed, they are fully determined by the memory region that is pointed to by the owned pointer. Note that `n` and `tail` are linked back to the multiset `s` using a `rc::constraints` annotation. Another such constraint is used to enforce that `n` is smaller or equal than any element of `tail`, which directly implies that the list of chunks is sorted by increasing size.

The last interesting point concerning the definition of `chunks_t` is the `rc::size` annotation on line 5. It specifies a number of bytes up to which the block should be padded. Indeed, **`struct`** `chunk` is used as a header for the memory chunk, but the intent of the type is to denote the whole chunk. As a consequence, the `rc::size` annotation adds the appropriate amount of padding and gives a type of uninitialized memory to the part of the chunk that is not meaningful.

Loop invariant and verification. Given the presentation of the annotations in the previous sections, the formal specification of `free` should not be too surprising. One point that could have easily been overlooked by a user though, is the fact that the function needs to have a precondition (line 14) stating that the block of memory that is being freed can fit a **`struct`** `chunk`.

For RefinedC to verify `free` it is necessary to provide an explicit loop invariant on line 19-21. Loop invariants are specified using annotations of three forms: existentially quantified parameters given with `rc::exists`, types for the involved local variables and function arguments given with `rc::inv_vars`, and constraints given with `rc::constraints`. (No such constraint is used for `free`.)

Intuitively, the loop invariant keeps track of the ownership of the list that is being traversed. There are two parts to it: the tail of the list that is held by local variable `cur`, and the prefix of the list that has already been explored. The former is expressed in a straightforward way using the two existentially quantified variables `cp` (the location that `cur` points to) and `cs` (the multiset represented by `cur`). The question is: where should the ownership of the prefix of the list be placed? The answer is that the prefix of the list is owned by the type of `list`, *i.e.*, the pointer to the full list. This is achieved using the `wand<...>` type on line 21, which, as its name suggests, is derived from the magic wand of separation logic. Intuitively, the type of `list` in the loop invariant allows to recover ownership of the full list (after insertion), given ownership of the suffix of the list (*i.e.*, the part stored in `cur`) to which the new chunk has been added.

Finally, RefinedC need a little bit of help to complete the correctness proof. The verification of `free` generates domain-specific side conditions (involving multisets) that cannot be discharged by RefinedC's default automation. The `rc::tactics` annotation is hence used on line 16 to instruct RefinedC to solve them using a domain-specific solver `multiset_solver` for multisets provided by the std++ library [The Coq-std++ Team 2020].

3 THE CAESIUM LANGUAGE

Before a C program can be verified using RefinedC (§4), it is elaborated by the RefinedC front end to a core language called **Caesium**. We discuss the syntax, operational semantics, and memory model of Caesium, the elaboration of C source code, and some design decisions and limitations.

$$
\begin{aligned}
\mathsf{alloc}(\mathsf{d}, \mathsf{size}) &\triangleq \texttt{if}\ \mathsf{use}(\mathsf{size}) > \mathsf{use}(\mathsf{load}(\mathsf{d}).\mathsf{len})\ \texttt{then}\ (\texttt{return}\ \mathtt{NULL})\ \texttt{else} \\
&\quad\ \mathsf{store}(\mathsf{load}(\mathsf{d}).\mathsf{len},\ \mathsf{use}(\mathsf{load}(\mathsf{d}).\mathsf{len}) - \mathsf{use}(\mathsf{size})); \\
&\quad\ \texttt{return}\ \mathsf{use}(\mathsf{load}(\mathsf{d}).\mathsf{buffer}) + \mathsf{use}(\mathsf{load}(\mathsf{d}).\mathsf{len}) \\
n_{\mathsf{size}}\ @\ \mathsf{alloc}_{\mathsf{data}} &\triangleq \mathsf{struct}\ [n_{\mathsf{size}}\ @\ \mathsf{int}(\mathsf{size\_t}),\ \&_{\mathsf{own}}(\mathsf{uninit}(n_{\mathsf{size}}))] \\
\mathsf{alloc}_{\mathsf{spec}} &\triangleq \mathsf{fn}(\forall(n_{\mathsf{len}}, n_{\mathsf{size}}, \mathsf{p}).\ \mathsf{p}\ @\ \&_{\mathsf{own}}(n_{\mathsf{len}}\ @\ \mathsf{alloc}_{\mathsf{data}}),\ n_{\mathsf{size}}\ @\ \mathsf{int}(\mathsf{size\_t}); \mathsf{True}) \\
&\quad \to \exists().\ (n_{\mathsf{size}} \le n_{\mathsf{len}})\ @\ \mathsf{optional}(\&_{\mathsf{own}}(\mathsf{uninit}(n_{\mathsf{size}})),\ \mathsf{null}); \\
&\qquad\ \mathsf{p} \triangleleft_l ((n_{\mathsf{size}} \le n_{\mathsf{len}})\ ?\ (n_{\mathsf{len}} - n_{\mathsf{size}}) : n_{\mathsf{len}})\ @\ \mathsf{alloc}_{\mathsf{data}}
\end{aligned}
$$

Fig. 6. The encoding of the `alloc` function from Figure 2 in Caesium and its specification in terms of the RefinedC type system (slightly simplified).

Syntax. The syntax of the RefinedC core language is as follows:

$$
\begin{aligned}
\mathit{LExpr} \ni p &::= \mathsf{i} \mid \ell \mid \mathsf{load}^{\mathsf{o}}_{\iota}(p) \mid p.{}_{\sigma}\mathsf{i} \mid p +^{\mathsf{p}}_{\iota} e \\
\mathit{RExpr} \ni e &::= \mathsf{i} \mid v \mid \odot_{\{\theta\}} e \mid e_1 \odot_{\{\theta_1,\theta_2\}} e_2 \mid \mathsf{use}_{\iota}(p) \mid \&p \mid \mathsf{annot}_x(e) \mid \ldots \\
\mathit{Stmt} \ni s &::= \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \mathsf{store}^{\mathsf{o}}_{\iota}(p, e);\ s \mid \texttt{let}\ \mathsf{i} = \texttt{call}\ e(\overline{e});\ s \mid \texttt{return}\ e \mid \mathsf{annot}_x s \\
&\quad \mid \texttt{goto}\ \mathsf{b} \mid \mathsf{switch}_{\iota}\ e\ \texttt{case}\ \overline{z \Rightarrow s}\ \texttt{default}\ s \mid \ldots
\end{aligned}
$$

We let $\mathsf{i} \in \mathrm{Name}$ range over variable names, $\ell \in \mathit{Loc}$ over location literals, $v \in \mathit{Val}$ over values, and $\mathsf{b} \in \mathit{BlockId}$ over block identifiers. The notation $\overline{x}$ denotes a list. To give an idea of what Caesium programs look like, the translation of the `alloc` function from Figure 2 is shown in Figure 6.

Caesium's expressions are pure, whereas its statements can have side effects. As usual for low-level languages, Caesium distinguishes between r-expressions, which result in a value, and l-expressions, which result in a location. Due to this distinction, there are two operations to read a value from memory: the l-expression $\mathsf{load}^{\mathsf{o}}_{\iota}(p)$ obtains the location stored at p, whereas the r-expression $\mathsf{use}_{\iota}(p)$ obtains the value stored at p. An l-expression p can be manipulated using the member operator $p.{}_{\sigma}\mathsf{i}$, which takes field i of a struct, and the offset operator $p +^{\mathsf{p}}_{\iota} e$, which increases its offset by e. Moreover, an l-expression can be converted into an r-expression that designates a pointer value using the address operator $\&p$.

The Caesium syntax contains some annotations that are used by the operational semantics to disambiguate operator overloading. Memory accesses (load, use, store) are annotated by a *layout* ι, which gives the number of bytes affected, and an *access order* o, which specifies if the operation is non-atomic or sequentially consistent. The member operator is annotated by the *struct layout* σ of its operand, the offset operator is annotated by the layout ι of its operand, and binary operators $\odot$ are annotated by the C types θ (*i.e.*, pointers or integers) of their operands. These annotations are automatically inferred by the RefinedC front end, and thus typically left implicit.

To handle all forms of control offered by C (loops, break, continue, goto, and unstructured switch), Caesium is control-flow graph based. Caesium programs are represented as maps from labels to statements, and the goto b statement can be used to jump between these blocks. Caesium's only branching construct is the switch statement $\mathsf{switch}_{\iota}\ e\ \texttt{case}\ \overline{z \Rightarrow s}\ \texttt{default}\ s$, which executes a branch $z \Rightarrow s$ depending on the resulting integer z of evaluating e, or the default s if no matching branch exists. The $\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2$ statement is defined using switch.

The statement $\mathsf{annot}_{\mathsf{Unwrap}}\&p;\ s$ and the expression $\mathsf{annot}_{\mathsf{Wrap}}(e)$ correspond to annotations in the C code like `rc_unwrap(p)` and `rc_wrap(e)`.

Operational semantics and memory model. Caesium is given a small-step operational reduction semantics. Following the usual design of languages embedded in Iris, the semantics is substitution based, but with a twist. To model that one can take the address of function arguments and local variables in C, they are stored via an indirection on the heap. When a function is invoked, its arguments and local variables are substituted by location literals that refer to their respective values. Since expressions do not involve control, their operational semantics is defined in a standard way using evaluation contexts. Due to control and function calls, the operational semantics of statements is more involved: it uses a stack to keep track of the functions that have been called. To handle concurrency, it uses a thread-pool-based semantics similar to other languages in Iris.

Caesium uses a low-level memory model that is roughly based on that of CompCert [Leroy and Blazy 2008; Leroy et al. 2012]. On top of that, it uses the semantics of RustBelt [Jung et al. 2018a] to assign undefined behavior to data-races on non-atomic memory accesses.

Elaboration of C source code by the* RefinedC *front end. To verify a program using RefinedC, the RefinedC front end elaborates C source files into ASTs in a deeply-embedded version of Caesium in Coq. The RefinedC front end is an OCaml program [Anonymous 2020] built on top of the first half of the pipeline of Cerberus [Memarian et al. 2019]. The Cerberus pipeline produces an intermediate representation close to source C, but annotated with type information and explicit conversions between l- and r-expressions. Programs in this intermediate representation are translated into Caesium by desugaring control flow constructs (including unstructured switch, like Duff's device) into gotos. Expressions with nested side-effects are desugared into statements.

Design choices and limitations. 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 from one another regarding many aspects of the C memory model [Wang et al. 2012; Memarian et al. 2016, 2019]. Hence, the Caesium memory model has less undefined behavior than ISO C with respect to, *e.g.*, padding in structs and effective types.

RefinedC lacks some features of ISO C that are subject to active research in program verification and separation logic. It does not support C's loose ordering of expression evaluation [Krebbers 2014; Frumin et al. 2019] (Caesium fixes a left-to-right ordering), lifetimes of block-scoped variables [Krebbers and Wiedijk 2013] (all local variables are function scoped in Caesium), and relaxed-memory concurrency [Batty et al. 2011; Kaiser et al. 2017; Dang et al. 2020] (Caesium supports sequentially-consistent and non-atomic accesses). To mitigate the first two points, the RefinedC front end performs an over-approximating analysis that emits warnings if it finds expressions that may be non-deterministic or allow the address of a block-scoped variable to escape.

4 THE REFINEDC TYPE SYSTEM

In this section we introduce the RefinedC type system using the `alloc` function of §2.1 (or rather, its translation to Caesium given in Figure 6) as a running example. We start by defining the notion of type used by RefinedC, and reviewing a number of interesting type constructors (§4.1). We then gradually introduce several judgments of the type system, as well as highlights of their semantic model, starting with typing judgments for statements (§4.2). We also show how the typing rules of RefinedC are expressed using separation logic (§4.3), and how the type system can be extended to support low-level programming idioms (§4.4). Finally, we discuss two key technical ideas that make RefinedC's approach possible: a meta-level subsumption judgment to glue reasoning steps together (§4.5), and the mechanism that supports fine-grained ownership changes (§4.6).

Type	Refinement	Intuitive semantics
$\mathsf{int}(\alpha)$	mathematical integer n	C integer of type α that encodes n
boolean	decidable proposition ϕ	C Boolean reflecting the truth of ϕ
$\&_{\mathsf{own}}(\tau)$	memory location ℓ	unique ownership of τ at location ℓ
$\mathsf{uninit}(n)$	-	n uninitialized (*i.e.*, arbitrary) bytes
ptr	memory location ℓ	pointer ℓ without ownership
null	-	singleton type of `NULL`
$\mathsf{optional}(\tau_1, \tau_2)$	decidable proposition ϕ	if ϕ then τ_1 else τ_2
$\mathsf{wand}(H, \tau)$	-	elements of τ with hole H
$\mathsf{struct}_\sigma\ \overline{\tau}$	-	struct with layout σ and fields of types $\overline{\tau}$
$\exists x.\ \tau(x)$	-	type-level existential quantifier
$\{\tau \mid \phi\}$	-	elements of τ satisfying proposition ϕ
$\mathsf{padded}(\tau, n)$	-	elements of τ padded to n bytes

Table 1. A selection of RefinedC types with their intuitive semantics.

4.1 Types

The first problem when building a type system for C—especially a refinement type system as noted by Rondon [2012]—is that C types do not give useful guarantees about runtime values. Indeed, they only describe physical operations and the memory layout of values. As a consequence, RefinedC is *not* built on top of the C type system, but defines its own types. C types are used only as a description of the memory layout of values.

Since we want extensibility, RefinedC is not built from a fixed set of syntactic type constructors. Instead, every type τ is defined semantically, as in RustBelt [Jung et al. 2018a]. New types can be added by defining their meaning. Types are *assigned* to locations and values by the judgments $\ell \triangleleft_l \tau$ (location ℓ has type τ) and $v \triangleleft_v \tau$ (value v has type τ).[9] For example, the type int(size_t) used in the second argument of `alloc` in Figure 2[10] is intended to represent a `size_t` integer, so the assignment $v \triangleleft_v$ int(size_t) states that the bytes of value v represent a `size_t`, and $\ell \triangleleft_l$ int(size_t) asserts that location ℓ stores a `size_t`.

In RefinedC a type can also have a *refinement*, an optional parameter that restricts values in the type. For example, the type int(size_t) can be refined by a mathematical integer n to form the type n @ int(size_t) that represents the *singleton* set $\{n\}$ of `size_t` integers. A refinement attached to a type always represents some logical property of values of the type, but the exact meta-level sort of the refinement and the exact logical property vary across refinement types. For example, the type `chunks_t` in §2.3 is refined by (and thus represented by) the multiset of chunks stored in the list.

Representative types. The intuitive semantics of the most interesting RefinedC types used in the examples of §2 is given in Table 1. (Of course, RefinedC offers many more type constructors in its library, and even more can be defined by the user.) Most of these types have a straightforward interpretation, so we only focus on key features. The type ϕ @ boolean represents a Boolean value equivalent to the validity of ϕ. As explained later, this type is used for checking `if` statements. The refinement type ℓ @ $\&_{\mathsf{own}}(\tau)$ denotes an *owned* (non-aliased) pointer and its refinement ℓ specifies the exact memory location that is owned. For example, the annotations on `alloc_data` on line 3 in Figure 2 use ℓ @ $\&_{\mathsf{own}}(\tau)$ together with $\mathsf{uninit}(n)$ to denote a pointer to a block of n bytes

[9] Additionally, every type τ has an associated size $\mathsf{sizeof}(\tau)$ indicating the number of bytes that its values occupy in memory.
[10] Note that we use a mathematical syntax that slightly differs form the concrete annotations. For example, int(size_t) is written `int<size_t>` in the concrete syntax. We leave the obvious syntax translation implicit.

of uninitialized memory. RefinedC also supports pointers without associated ownership (written ℓ @ ptr), but they are used rarely. The type ϕ @ optional(τ_1, τ_2) encodes a case distinction on the validity of ϕ at the type level, and it is most commonly used to represent nullable pointers (via $\&_{\mathsf{own}}(\tau)$ and null), as seen in §2. Another interesting type is wand(H, τ), which is used to encode partial data structures via the magic wand as described by [Cao et al. 2019]. It is used in the loop invariant of `free` in Figure 5 on line 21 to represent the list segment that has already been processed by the loop.

Although they can be used directly, the four types that appear at the bottom of Table 1 are most often generated from annotations. For instance, a structure type $\mathsf{struct}_\sigma\ \overline{\tau}$ is constructed by combining the types given by the `rc::field` annotations (see line 2-3 in Figure 2) on a C `struct`. Similarly, the types $\exists x.\ \tau(x)$ and $\{\tau \mid \phi\}$ encode the `rc::exists` and `rc::constraints` annotations (see line 4 and 6 of Figure 5). Finally, the padded(τ, n) type, representing a block of n bytes with a header of type τ, is generated from the `rc::size` annotation (see line 5 of Figure 5).

Function pointer type. There is one important RefinedC type that was omitted in the discussion above: the type of functions. The function pointer type

$$\mathsf{fn}(\forall x.\ \overline{\tau_{\mathsf{arg}}}; H_{\mathsf{pre}}) \to \exists y.\ \tau_{\mathsf{ret}}; H_{\mathsf{post}}$$

plays a crucial role as it carries the specification of the function. Showing that the implementation of a function inhabits this type is the main lemma that is automatically generated and proven for each annotated function in the source code. The different parts of the function type are derived from the source code annotations. For example, the annotations on `alloc` (line 6-9 of Figure 2) lead to the definition of the type $\mathsf{alloc}_{\mathsf{spec}}$ given in Figure 6. In particular, the variables given in the `rc::parameters` annotation (line 6) are bundled into a tuple and bound by x in the whole function type. The type of the function arguments and return value given by `rc::args` (line 7) and `rc::returns` (line 8) appear respectively as $\overline{\tau_{\mathsf{arg}}}$ and τ_{ret}. The pre- and postconditions given by `rc::requires` and `rc::ensures` (line 9) appear as H_{pre} and H_{post} respectively. Existential variables that are bound in the return type and the postconditions can be given using `rc::exists` and correspond to the variable y.

4.2 Introduction to Type Checking in RefinedC

Having introduced RefinedC types and the two forms of type assignments, we now turn to the type system and its typing judgments.

From specification to statement judgment. Before looking at RefinedC's typing judgments and typing rules, we need to understand how the RefinedC type system works in general. We illustrate this using the running example of the `alloc` function from Figure 2. Its specification described in Figure 6 is equivalent to the following meta-level entailment:

$$\ell_{\mathsf{d}} \lhd_l \mathsf{p}\ @\ \&_{\mathsf{own}}(n_{\mathsf{len}}\ @\ \mathsf{alloc}_{\mathsf{data}}),\ \ell_{\mathsf{size}} \lhd_l n_{\mathsf{size}}\ @\ \mathsf{int}(\mathsf{size\_t}) \Vdash \vdash^{\Sigma}_{\mathrm{STMT}} \mathsf{alloc}(\ell_{\mathsf{d}}, \ell_{\mathsf{size}}) \quad (\textsc{alloc-entail})$$

RefinedC's judgments (like $\vdash^{\Sigma}_{\mathrm{STMT}}$) do not have an explicit context. Instead, type assignments are stored in the *meta-level context* (the left-hand side of the meta-level entailment $\Vdash$). This is similar to how type systems are encoded in logical frameworks [Harper et al. 1993; Pfenning and Schürmann 1999]. In this example, type checking of `alloc` assumes that the locations ℓ_{d} and ℓ_{size} storing the arguments have the types specified by `rc::args`.

Typing judgments for statements. We now explain the judgment $\vdash^{\Sigma}_{\mathrm{STMT}} \mathsf{alloc}(\ell_{\mathsf{d}}, \ell_{\mathsf{size}})$, which appears in the conclusion of the entailment ALLOC-ENTAIL. Intuitively, $\vdash^{\Sigma}_{\mathrm{STMT}}\ s$ asserts that the

statement s is well typed and, hence, safe to execute.[11] The judgment is parametrized by a *function state* $\Sigma = (C, \overline{(\ell, n)}, \exists x.\, \tau(x); H(x))$ containing three components. The first one is the control-flow graph C of the function (it is trivial for `alloc` since it only has one block). The second component is a list $\overline{(\ell, n)}$ recording the locations that are allocated on the function's stack frame (*i.e.*, the arguments and the local variables) together with their memory layouts. The last component $\exists x.\, \tau(x); H(x)$ encodes the return type $\tau(x)$ (`rc::returns`) and postcondition $H(x)$ (`rc::ensures`), both of which can be optionally parametrized by an existentially quantified variable x (`rc::exists`, not used by the examples in §2). In §4.5, we explain the last two components' role in verifying the `return` statement.

Typing rules for $\vdash^{\Sigma}_{\text{STMT}}$ and $\vdash_{\text{EXPR}}$. Next, we show some typing rules that we use for type checking `alloc`. We start with the T-IF rule, since the first statement of alloc($\ell_{\text{d}}, \ell_{\text{size}}$) is a conditional:

$$\text{T-IF}\ \frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ \vdash^{\Sigma}_{\text{IF}} v : \tau\ \texttt{then}\ s_1\ \texttt{else}\ s_2\}}{\vdash^{\Sigma}_{\text{STMT}}\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

The premise of T-IF relies on the judgments $\vdash_{\text{EXPR}} e\ \{v, \tau.\ G(v, \tau)\}$ and $\vdash^{\Sigma}_{\text{IF}} v : \tau\ \texttt{then}\ s_1\ \texttt{else}\ s_2$. As for statements, the typing judgment of expressions $\vdash_{\text{EXPR}} e\ \{v, \tau.\ G(v, \tau)\}$ asserts that its subject (expression e) can be safely executed. However, unlike statements, expressions produce a value, hence the expression typing judgment contains a continuation G (similar to the postcondition of the weakest precondition assertion of Iris)[12] in which the variables v and τ are bound. Intuitively, v represents the value that e evaluates to, and τ is the type of v that was inferred by $\vdash_{\text{EXPR}}$. The specialized judgment $\vdash^{\Sigma}_{\text{IF}}$ appearing in the continuation is used to type check `if`-statements. Before explaining the purpose of these specialized judgments, we continue with type checking alloc($\ell_{\text{d}}, \ell_{\text{size}}$). The next step is checking the binary operator > in (`use`(size) > `use`(`load`(d).len)) with the rule

$$\text{T-BINOP}\ \frac{\vdash_{\text{EXPR}} e_1\ \{v_1, \tau_1.\ \vdash_{\text{EXPR}} e_2\ \{v_2, \tau_2.\ \vdash_{\text{BINOP}} v_1 : \tau_1 \odot v_2 : \tau_2\ \{v_3, \tau_3.\ G(v_3, \tau_3)\}\}\}}{\vdash_{\text{EXPR}} e_1 \odot e_2\ \{v, \tau.\ G(v, \tau)\}}$$

Similar to T-IF, rule T-BINOP first checks the constituent expressions using $\vdash_{\text{EXPR}}$ and then introduces a specialized judgment $\vdash_{\text{BINOP}}$ for checking the binary operation $\odot$ (> in the example). Skipping over the reads of the operands of >, the type checker arrives at the following judgment:

$$\vdash_{\text{BINOP}} v_1 : n_{\text{size}}\ @\ \mathsf{int}(\mathsf{size\_t}) > v_2 : n_{\mathsf{len}}\ @\ \mathsf{int}(\mathsf{size\_t})\ \{v_3, \tau_3.\ \vdash^{\Sigma}_{\text{IF}} v_3 : \tau_3\ \texttt{then}\ s_1\ \texttt{else}\ s_2\}\quad (\text{ALLOC-GT})$$

This concludes the discussion of the typing rules of $\vdash^{\Sigma}_{\text{STMT}}$ and $\vdash_{\text{EXPR}}$[13], whose purpose is to traverse the program and delegate the meat of type checking to specialized judgments like $\vdash^{\Sigma}_{\text{IF}}$ and $\vdash_{\text{BINOP}}$.

Specialized typing judgments. The distinction between $\vdash^{\Sigma}_{\text{STMT}}$ and $\vdash_{\text{EXPR}}$ and the specialized judgments is necessary since selecting the right typing rule depends on the type of the operands, but this information first needs to be inferred using the typing rules for $\vdash^{\Sigma}_{\text{STMT}}$ and $\vdash_{\text{EXPR}}$. Thus, RefinedC has a specialized typing judgment for each operation whose type checking depends on types it operates on. The full list of judgments can be found in Appendix B, but here we focus on $\vdash^{\Sigma}_{\text{IF}}$ and $\vdash_{\text{BINOP}}$. Since the number of typing rules of RefinedC is quite intimidating (in the order of hundreds), we only show the rules relevant for our example, and refer the interested reader to the accompanying Coq development [Anonymous 2020], which also contains the soundness proof of all rules.

[11] This judgment is encoded using Iris's connective for weakest preconditions as shown in Appendix B. Iris' adequacy theorem ensures safety of the program.

[12] This is not a coincidence as $\vdash_{\text{EXPR}}$ is defined using Iris weakest preconditions (see Appendix B).

[13] The full set of typing rules for $\vdash^{\Sigma}_{\text{STMT}}$ and $\vdash_{\text{EXPR}}$ can be found in Appendix A.

Continuing with the running example (ALLOC-GT), the first typing we encounter is the following:

$$\text{O-GT}\ \frac{G(n_1 > n_2, (n_1 > n_2)\ @\ \text{boolean})}{\vdash_{\text{BINOP}}\ v_1 : n_1\ @\ \text{int}(\alpha) > v_2 : n_2\ @\ \text{int}(\alpha)\ \{v_3, \tau_3.\ G(v_3, \tau_3)\}}$$

As this rule shows, type checking > is straightforward: the operation results in a Boolean value corresponding to the truth of the mathematical proposition $n_1 > n_2$, and with the inferred type $(n_1 > n_2)$ @ boolean. Application of O-GT to ALLOC-GT results in the new goal

$$\vdash^{\Sigma}_{\text{IF}}\ n_{\text{size}} > n_{\text{len}} : (n_{\text{size}} > n_{\text{len}})\ @\ \text{boolean}\ \texttt{then}\ (\texttt{return}\ \texttt{NULL})\ \texttt{else}\ \ldots \qquad \text{(ALLOC-IF)}$$

4.3 Formulating Typing Rules using Separation Logic

So far, we have shown some typing rules, but not explained their *general* syntax. The general syntax is important since the RefinedC type system can be extended with new rules. The general form of RefinedC typing rules is:

$$\frac{G \quad \text{(premise)}}{F \quad \text{(judgment)}} \quad \psi\ \text{(optional side condition)}$$

Here, F is a typing judgment, and ψ contains additional restrictions on when the rule applies.

The interesting question is what form G has. On one hand, G needs to be expressive enough to encode interesting typing rules. On the other hand, G must have some logical interpretation so that we can meaningfully prove the rule sound. The natural choice that fulfills both requirements is to identify G with formulas of *separation logic*, which has proven to be a powerful tool for verifying low-level programs [Chlipala 2011; Cao et al. 2018; Jung et al. 2018a,b].[14]

To illustrate how this works, we return to ALLOC-IF. Here we need to type check an `if`-statement on a value of type ϕ @ boolean. Intuitively, this should result in two branches for `then` and `else`: In the `then` branch, we can assume ϕ, whereas in the `else` branch, we can assume $\neg\phi$. The following rule shows how this reasoning is encoded in separation logic:

$$\text{IF-BOOL}\ \frac{(\ulcorner\phi\urcorner \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}}\ s_1) \wedge (\ulcorner\neg\phi\urcorner \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}}\ s_1)}{\vdash^{\Sigma}_{\text{IF}}\ v : \phi\ @\ \text{boolean}\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

This rule uses the (normal) conjunction $\wedge$ to represent the two branches, and the magic wand to introduce a local premise in each (here, the pure propositions ϕ resp. $\neg\phi$, which are embedded into separation logic via $\ulcorner\cdot\urcorner$).[15] Applying IF-BOOL in ALLOC-IF leads to the two expected subgoals:

$$\ulcorner n_{\text{size}} > n_{\text{len}}\urcorner \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}}\ \texttt{return}\ \texttt{NULL} \qquad\qquad \ulcorner\neg(n_{\text{size}} > n_{\text{len}})\urcorner \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}}\ \ldots$$

Here, `return NULL` is the `then`-branch of alloc, and ... is the `else`-branch. (More examples of encoding RefinedC typing rules in separation logic appear in later sections.)

4.4 Using Extensibility to Encode Low-Level C Idioms

In this section, we explain one of the key benefits of representing typing rules in separation logic as explained above. In idiomatic C code, programmers often use the same physical operation in many different contexts and with very different semantic intent. Thus, to match the intuition of the programmer, it is necessary that the same physical operation lead to different updates on the logical state during verification. RefinedC enables this by overloading typing rules based on the

[14] Readers unfamiliar with separation logic can read the magic wand $\mathrel{-\!\!*}$ as an implication and the separating conjunction $*$ as a normal conjunction, except that the ownership used to prove both sides must be disjoint.

[15] A reader well-versed in separation logic might wonder why we use the magic wand instead of implication. Both versions are equivalent since ϕ is pure. We follow the Iris style, which prefers magic wand over implication.

RefinedC types they operate on. This allows creating specialized typing rules for specific low-level C idioms, which is crucial to achieving the low annotation overhead witnessed in §2.

Specialized rule for the offset operation. To illustrate the overloading of typing rules, we consider the offset operation on pointers. In most verification tools based on symbolic execution, when an offset operation is verified, only its result is recorded. In particular, there is no additional change to the logical state. In RefinedC this is also the case in the following rule O-ADD-PTR for the *raw pointer* type ptr—the type of pointers that do not have associated ownership:

$$\text{O-ADD-PTR}\ \frac{G(\ell +_l n_2, (\ell +_l n_2)\ @\ \mathsf{ptr})}{\vdash_{\text{BINOP}} v_1 : \ell\ @\ \mathsf{ptr} + v_2 : n_2\ @\ \mathsf{int}(\mathsf{size\_t})\ \{v_3, \tau_3.\ G(v_3, \tau_3)\}}$$

Here, $+_l$ is the offset operator on the mathematical representation of locations.

However, it is often the case that the programmer implicitly intends to modify the logical state, in particular, ownership, when performing a pointer offset. For example, the expression `d->buffer + d->len` of the `alloc` function (on line 13 of Figure 2) intuitively splits the ownership of `d->buffer` into two pieces: a first part that remains associated with `d->buffer`, and a second part that is eventually returned to the caller as the allocated block of memory. In RefinedC, this intuitive reasoning can be formally encoded using the following rule O-ADD-UNINIT:

$$\text{O-ADD-UNINIT}\ \frac{\ulcorner 0 \le n_2 \le n_1 \urcorner * (v_1 \lhd_v \ell\ @\ \&_{\mathsf{own}}(\mathsf{uninit}(n_2)) \mathrel{-\!\!*} G(\ell +_l n_2, (\ell +_l n_2)\ @\ \&_{\mathsf{own}}(\mathsf{uninit}(n_1 - n_2))))}{\vdash_{\text{BINOP}} v_1 : \ell\ @\ \&_{\mathsf{own}}(\mathsf{uninit}(n_1)) + v_2 : n_2\ @\ \mathsf{int}(\mathsf{size\_t})\ \{v_3, \tau_3.\ G(v_3, \tau_3)\}}$$

Back to the example. Instead of explaining O-ADD-UNINIT directly, let us see the rule in action in the context of type checking `alloc`, for which the following judgment must be proved.[16]

$$\vdash_{\text{BINOP}} v_b : \ell\ @\ \&_{\mathsf{own}}(\mathsf{uninit}(n_{\mathsf{len}})) + v_2 : (n_{\mathsf{len}} - n_{\mathsf{size}})\ @\ \mathsf{int}(\mathsf{size\_t})\ \{v, \tau.\ G(v, \tau)\} \qquad \text{(ALLOC-ADD)}$$

To apply O-ADD-UNINIT, we first need to check that the offset n_2 (here $n_{\mathsf{len}} - n_{\mathsf{size}}$) stays in bounds of the block of uninitialized memory of size n_1 (here n_{len}). Then, O-ADD-UNINIT can split the block into the two pieces: the original block at location ℓ shrinks to only n_2 bytes (here $n_{\mathsf{len}} - n_{\mathsf{size}}$), and its ownership is passed to the continuation, whereas the returned pointer at $\ell +_l n_2$ receives ownership of $n_1 - n_2$ bytes (here $n_{\mathsf{len}} - (n_{\mathsf{len}} - n_{\mathsf{size}}) = n_{\mathsf{size}}$). Concretely, after the application of O-ADD-UNINIT the ownership of $v_b \lhd_v \&_{\mathsf{own}}(\mathsf{uninit}(n_{\mathsf{len}}))$ is split into the following disjoint parts:[17]

$$v_b \lhd_v \&_{\mathsf{own}}(\mathsf{uninit}(n_{\mathsf{len}} - n_{\mathsf{size}})) \qquad v_r \lhd_v \&_{\mathsf{own}}(\mathsf{uninit}(n_{\mathsf{size}})) \qquad \text{(ALLOC-ADD-AFTER)}$$

The first part is used to establish the postcondition of `alloc` (as discussed in §4.6), while the second part is returned to the caller, which exactly matches the implicit intent of the programmer.

In summary, we have seen how typing rules like O-ADD-UNINIT can encode idioms found in low-level C code, matching the intuitive reasoning of the programmer. The key insight is to have multiple typing rules for the same physical operation that update the logical state (of the verifier) differently depending on the types of the operands. We also note that the rules O-ADD-PTR and O-ADD-UNINIT do not cover all possible programming idioms involving pointer arithmetic. However, new rules can be added to support new idioms, relying on RefinedC's extensible rule set.

The handling of the ownership of `d->buffer` during type checking, of which we already saw a part in this section, serves as a running example in the following two sections (§4.5–§4.6). As a reference, Figure 7 summarizes the different steps of ownership transfer for our running example, with the corresponding typing rules and varying type assignments in the context.

[16] Details of how we reach this judgment are explained in §4.6.

[17] We ignore the refinement ℓ as it is irrelevant for the rest of the type checking.

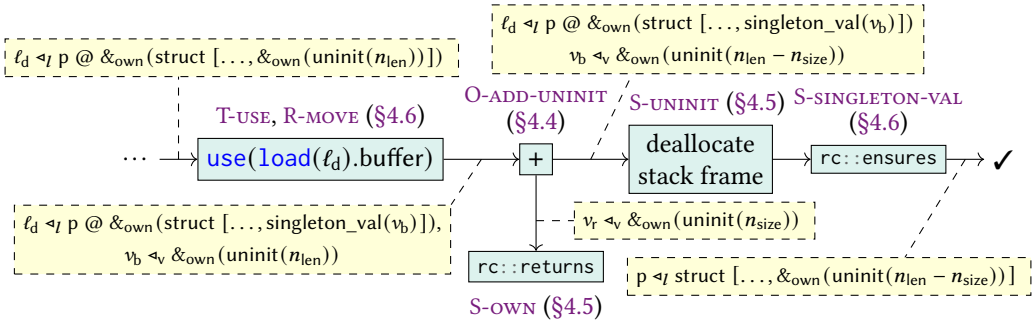

Fig. 7. An illustration how the ownership of `d->buffer` flows through the verification of `alloc`. The solid boxes show the different stages of the verification with corresponding typing rules. The dashed boxes show how the type assignments in the context evolve during the verification.

4.5 Reconciling Type Assignments via Subsumption

We now turn to an important meta-level rule for *subsumption* of type assignments. Primarily, the need for subsumption arises when a rule requires a variable to have a certain type, but the context contains a different type for it. A common case is type checking a `return` statement with the following rule:

$$\text{T-RETURN}\ \frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ \exists x.\ v \triangleleft_v \tau(x) * \overline{\ell \triangleleft_l \mathsf{uninit}(n)} * H(x)\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{return}\ e}\ \Sigma = (C,\ \overline{(\ell, n)},\ \exists x.\ \tau(x); H(x))$$

Type checking (`return` e) consists of the following steps: (1) type checking e using the $\vdash_{\text{EXPR}}$ judgment, (2) finding an instantiation for the existentially quantified variables x in the return type and postcondition (as specified by `rc::exists`), (3) showing that the returned value v has the expected return type $\tau(x)$ (as specified by `rc::returns`) by establishing $v \triangleleft_v \tau(x)$, (4) giving up ownership $\overline{\ell \triangleleft_l \mathsf{uninit}(n)}$ of the stack frame, and (5) proving the postcondition $H(x)$ (as specified by `rc::ensures`).

The problem we want to illustrate arises in step (3). For example, when type checking the `return` in the `then` branch of `alloc` on line 11 of Figure 2, the type system infers the type null for the returned value `NULL`, but the `rc::returns` clause specifies an optional(τ_1, τ_2) type. This means that the type checker needs to resolve the following goal:

$$\ldots, v \triangleleft_v \mathsf{null} \Vdash\ v \triangleleft_v (n_{size} \le n_{len})\ @\ \mathsf{optional}(\&_{own}(\mathsf{uninit}(n_{size}))) * G \qquad \text{(ALLOC-NULL)}$$

Here, G contains the `rc::ensures` clause of `alloc`'s specification. Recall from §4.2 that $\Vdash$ is the meta-level entailment. This goal cannot be directly handled by RefinedC typing rules since they only apply to judgments, while the goal here is not a judgment, but a separating conjunction. Instead, such goals are handled via the following *meta-level* rules for subsumption:

$$\text{<:-E-VAL}\ \frac{v \triangleleft_v \tau_2 \in \Delta \quad \Delta \setminus \{v \triangleleft_v \tau_2\} \Vdash v \triangleleft_v \tau_2 <: v \triangleleft_v \tau_1\ \{G\}}{\Delta \Vdash v \triangleleft_v \tau_1 * G} \qquad \text{<:-E-LOC}\ \frac{\ell \triangleleft_l \tau_2 \in \Delta \quad \Delta \setminus \{\ell \triangleleft_l \tau_2\} \Vdash \ell \triangleleft_l \tau_2 <: \ell \triangleleft_l \tau_1\ \{G\}}{\Delta \Vdash \ell \triangleleft_l \tau_1 * G}$$

To check that a value v has type τ_1, rule <:-E-VAL searches the meta-level context Δ for a type assignment of v, and then introduces the *subsumption judgment* $A_1 <: A_2\ \{G\}$ between the two type assignment of v (Rule <:-E-LOC is similar but it applies to location assignment). Ignoring the role of the continuation G in this judgment for now, applying <:-E-VAL to ALLOC-NULL reduces the

goal to:

$$\ldots \Vdash\ v \lhd_\mathsf{v} \mathsf{null} <: v \lhd_\mathsf{v} (n_\mathsf{size} \le n_\mathsf{len})\ @\ \mathsf{optional}(\&_\mathsf{own}(\mathsf{uninit}(n_\mathsf{size})))\ \{G\}$$

This goal is resolved using S-NULL, which reduces the subsumption to $\ulcorner \neg(n_\mathsf{size} \le n_\mathsf{len}) \urcorner * G$:[18]

$$\text{S-NULL}\ \frac{\ulcorner \neg\phi \urcorner * G}{v \lhd_\mathsf{v} \mathsf{null} <: v \lhd_\mathsf{v} \phi\ @\ \mathsf{optional}(\&_\mathsf{own}(\tau), \mathsf{null})\ \{G\}}$$

Symmetrically, S-OWN applies to the `return` in the `else` branch on line 13 of Figure 2:

$$\text{S-OWN}\ \frac{\ulcorner \phi \urcorner * \&_\mathsf{own}(\tau_1) <: \&_\mathsf{own}(\tau_2)\ \{G\}}{v \lhd_\mathsf{v} \&_\mathsf{own}(\tau_1) <: v \lhd_\mathsf{v} \phi\ @\ \mathsf{optional}(\&_\mathsf{own}(\tau_2), \mathsf{null})\ \{G\}}$$

Subsumption without losing ownership. Since subsumption shows type inclusion (not equivalence), subsumption rules, in general, may result in the loss of ownership information. However, this ownership might be necessary to type check the rest of the program. For example, consider the following "bad" rule that we do not actually use in RefinedC:

$$\text{S-UNINIT-BAD}\ \frac{\ulcorner \mathsf{sizeof}(\tau) = n \urcorner * G}{\ell \lhd_l \tau <: \ell \lhd_l \mathsf{uninit}(n)\ \{G\}}$$

A subsumption rule for uninit is necessary to type check the deallocation of the stack frame in step (4) of T-RETURN. While S-UNINIT-BAD is sound, and is suitable for showing $\ell \lhd_l \mathsf{uninit}(n)$, it discards the ownership $v \lhd_\mathsf{v} \tau$ held by the value v that ℓ refers to. For example, applying S-UNINIT-BAD on line 13 of Figure 2 with $\ell = \ell_\mathsf{d}$ and $\tau = \mathsf{p}\ @\ \&_\mathsf{own}(\ldots)$ makes it impossible to prove the postcondition of `alloc`, as the ownership of $\mathsf{p}\ @\ \&_\mathsf{own}(\ldots)$ is irretrievably lost.

To address this problem, our subsumption judgment (like most other judgments in RefinedC) is equipped with a continuation G, to which one can pass ownership that was not consumed while proving the judgment. In particular, the following rule S-UNINIT resolves the problem of S-UNINIT-BAD by passing $v \lhd_\mathsf{v} \tau$ to the continuation G:

$$\text{S-UNINIT}\ \frac{\ulcorner \mathsf{sizeof}(\tau) = n \urcorner * (\forall v.\ v \lhd_\mathsf{v} \tau \mathrel{-\!\!*} G)}{\ell \lhd_l \tau <: \ell \lhd_l \mathsf{uninit}(n)\ \{G\}}$$

In our running example this means that $v \lhd_\mathsf{v} \mathsf{p}\ @\ \&_\mathsf{own}(\ldots)$ is available when type checking the `rc::ensures` clause, which, as shown in §4.6, is crucial to completing the proof.

As an aside, one might think that $A_1 <: A_2\ \{G\}$ is modeled (in Iris) as $(A_1 \mathrel{-\!\!*} A_2) * G$, but this encoding does not admit S-UNINIT because it does not allow ownership to be transferred from A_1 to G. Instead, to make rules like S-UNINIT provable, RefinedC encodes $A_1 <: A_2\ \{G\}$ as $A_1 \mathrel{-\!\!*} (A_2 * G)$.

4.6 Movement of Ownership

We now turn to the last missing pieces of the flow of ownership of `d->buffer` illustrated in Figure 7. In particular, since the exclusive ownership of $\&_\mathsf{own}$ cannot be copied, we describe how type checking the *read* of `d->buffer` on line 13 instead *moves* the ownership out of $\mathsf{alloc}_\mathsf{data}$. As a consequence, proving the postcondition requires moving the ownership of `d->buffer` back into $\mathsf{alloc}_\mathsf{data}$. We show how RefinedC handles such movement of ownership without user annotations.

[18] RefinedC actually uses more general rules, which are not specialized to $\&_\mathsf{own}$ and null.

Moving ownership out. To discuss this problem in detail, we first need to understand how the `use`(p) instruction, which reads from the l-expression p, is type checked in RefinedC. While the actual rule used by RefinedC can handle arbitrary l-expressions p (see Appendix A), we only show the rule for $p = \ell$ here, as this suffices to make our key point:

$$\text{T-USE}\ \frac{\exists \tau_1.\ \ell \triangleleft_l \tau_1 * \vdash_{\text{READ}} \tau_1\ \{v_2, \tau', \tau_2.\ \ell \triangleleft_l \tau' \mathrel{-\!\!*} G(v_2, \tau_2)\}}{\vdash_{\text{EXPR}} \texttt{use}(\ell)\ \{v, \tau.\ G(v, \tau)\}}$$

As with many of the typing rules for expressions, after finding a type τ_1 for ℓ, T-USE delegates type checking of the read to a specialized $\vdash_{\text{READ}}$ judgment. The $\vdash_{\text{READ}}$ judgment says that type checking a read from a location of type τ_1 produces a value v_2 with type τ_2 and, at the same time, it *changes* the type of the read location ℓ to τ'.

We explain below why reading sometimes changes the type, but we first consider the simpler case of reading an integer (*e.g.*, `size` on line 11 of Figure 2). Since integers do not contain exclusive ownership, reading an integer behaves as it would in a type system without ownership and thus does not change the type of the original location. Formally, types like $\mathsf{int}(\alpha)$ whose value assignments can be freely duplicated are called *copyable* (similar to the `Copy` trait in Rust [The Rust Team 2020]). Reads from copyable types are type checked using R-COPY:

$$\text{R-COPY}\ \frac{\forall v.\ G(v, \tau, \tau)}{\vdash_{\text{READ}} \tau\ \{v_2, \tau', \tau_2.\ G(v_2, \tau', \tau_2)\}}\ \tau \text{ copyable} \qquad \text{R-MOVE}\ \frac{\forall v.\ G(v, \mathsf{singleton\_val}(v), \tau)}{\vdash_{\text{READ}} \tau\ \{v_2, \tau', \tau_2.\ G(v_2, \tau', \tau_2)\}}$$

The more interesting case occurs when τ contains non-duplicable ownership, *i.e.*, τ is not copyable. In our running example, this happens when reading `d->buffer` on line 13 of Figure 2. Here, we must find a typing rule for $\vdash_{\text{READ}} \&_{\mathsf{own}}(\mathsf{uninit}(n_{\mathsf{len}}))\ \{v_2, \tau', \tau_2.\ G(v_2, \tau', \tau_2)\}$. The rule R-COPY does not apply since $\&_{\mathsf{own}}(\tau)$ contains exclusive ownership, and is thus not copyable. Instead, to type check a read from a location typed $\&_{\mathsf{own}}(\tau)$ we need to *move* the ownership of $\&_{\mathsf{own}}(\tau)$ out of the original location into the resulting value, as formalized by rule R-MOVE. This rule changes the type of the read location to $\mathsf{singleton\_val}(v)$, which is RefinedC's singleton type of the value v.[19] This singleton type acts as a marker which is necessary for moving ownership back in, as we will soon see. To come back to the example of reading `d->buffer`, T-USE[20] and R-MOVE create the following two type assignments:

$$\ell_{\mathsf{d}} \triangleleft_l \mathsf{p}\ @\ \&_{\mathsf{own}}(\mathsf{struct}\ [(n_{\mathsf{len}} - n_{\mathsf{size}})\ @\ \mathsf{int}, \mathsf{singleton\_val}(v_{\mathsf{b}})]) \qquad v_{\mathsf{b}} \triangleleft_{\mathsf{v}} \&_{\mathsf{own}}(\mathsf{uninit}(n_{\mathsf{len}}))$$

Here, the type assignment for ℓ_{d} remains in the typing context, whereas the type assignment of v_{b} is returned from the read, and is used by ALLOC-ADD on page 17.

Moving ownership in. So far we have seen how ownership can be moved out of types, leaving behind the marker $\mathsf{singleton\_val}(v)$, but it is equally important to move ownership back in and replace $\mathsf{singleton\_val}(v)$ with a more informative type. In our example, this happens when proving the postcondition of the function. Concretely, type checking the postcondition involves proving the following subsumption judgment:

$$v_{\mathsf{b}} \triangleleft_{\mathsf{v}} \&_{\mathsf{own}}(\mathsf{uninit}(n_{\mathsf{len}} - n_{\mathsf{size}})) \Vdash \mathsf{p} \triangleleft_l \mathsf{struct}\ [(n_{\mathsf{len}} - n_{\mathsf{size}})\ @\ \mathsf{int}, \mathsf{singleton\_val}(v_{\mathsf{b}})] <: \mathsf{p} \triangleleft_l \mathsf{struct}\ [(n_{\mathsf{len}} - n_{\mathsf{size}})\ @\ \mathsf{int}, \&_{\mathsf{own}}(\mathsf{uninit}(n_{\mathsf{len}} - n_{\mathsf{size}}))]\ \{\mathsf{True}\}$$

Here, the first type assignment of p is immediately derived from what is left over from ℓ_{d} after S-UNINIT (see §4.5). The second type assignment is the postcondition $\mathsf{p} \triangleleft_l (n_{\mathsf{len}} - n_{\mathsf{size}})\ @\ \mathsf{alloc}_{\mathsf{data}}$

[19] This is inspired by Alias Types [Smith et al. 2000] and Mezzo [Pottier and Protzenko 2013].
[20] Actually, we need the stronger T-USE from Appendix A that works for arbitrary l-expressions.

(as specified by `rc::ensures`) with the definition of $\mathsf{alloc}_{\mathsf{data}}$ unfolded and the conditional simplified. The type of v_b in the context is the result of O-ADD-UNINIT in ALLOC-ADD-AFTER on page 17 (see also Figure 7). After applying trivial subsumption rules, one ends up with

$$v_\mathsf{b} \lhd_\mathsf{v} \&_\mathsf{own}(\mathsf{uninit}(n_\mathsf{len} - n_\mathsf{size})) \Vdash \ell \lhd_l \mathsf{singleton\_val}(v_\mathsf{b}) <: \ell \lhd_l \&_\mathsf{own}(\mathsf{uninit}(n_\mathsf{len} - n_\mathsf{size}))\ \{\mathsf{True}\}$$

At this point, the ownership of v_b can be moved back into ℓ with the following subsumption rule:

$$\text{S-SINGLETON-VAL}\ \frac{\exists \tau'.\, v \lhd_\mathsf{v} \tau' * (\ell \lhd_l \tau' <: \ell \lhd_l \tau\ \{G\})}{\ell \lhd_l \mathsf{singleton\_val}(v) <: \ell \lhd_l \tau\ \{G\}}$$

Applying S-SINGLETON-VAL results in the following subsumption, which holds by reflexivity:

$$\Vdash \ell \lhd_l \&_\mathsf{own}(\mathsf{uninit}(n_\mathsf{len} - n_\mathsf{size})) <: \ell \lhd_l \&_\mathsf{own}(\mathsf{uninit}(n_\mathsf{len} - n_\mathsf{size}))\ \{\mathsf{True}\}$$

In summary, we have seen how R-MOVE can split one type assignment into two type assignments connected by $\mathsf{singleton\_val}(v)$, and how S-SINGLETON-VAL is used to merge the parts back together when necessary. It is important to point out that all of this splitting and merging of types is guided by the RefinedC type system and its rules. This significantly simplifies verification, and allows the RefinedC type system to automatically verify `alloc` in Figure 2 without additional annotations.

5 THE LITHIUM LOGIC PROGRAMMING LANGUAGE

In the previous section we saw that RefinedC's typing rules reduce judgments to separation logic formulas. In this section, we explain how we automate the search for typing derivations for such a separation logic-based type system. Our automation is based on the observation that the entire RefinedC type system can be represented in a *small fragment* of separation logic, which is amenable to *goal-directed* search. Goal-directed search, as the name suggests, works by picking rules based on the formula to be established (the goal). Goal-directed search is efficient and easy to implement. It forms the backbone of many logic programming languages like Lolli [Hodas and Miller 1994]. Our fragment of logic is, in fact, so restricted that we do not even need to *backtrack* during the search. This makes the search even faster. We call our fragment of separation logic **Lithium**. In the following, we describe the syntax of Lithium's formulas, how goal-directed search works in Lithium, why it does not require backtracking, and how we implement the search in Coq.

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:

Atoms	$A ::= v \lhd_\mathsf{v} \tau \mid \ell \lhd_l \tau \mid \ldots$
Basic goals	$F ::= \vdash^{\Sigma}_{\mathsf{STMT}} s \mid A_1 <: A_2\ \{G\} \mid \ldots$
Goals	$G ::= \mathsf{True} \mid F \mid H * G \mid H \mathrel{-\!\!*} G \mid G_1 \wedge G_2 \mid \forall x.\, G(x) \mid \exists x.\, G(x)$
Left-goals	$H ::= \ulcorner \phi \urcorner \mid A \mid H * H \mid \exists x.\, H(x)$
Unrestricted contexts	$\Gamma ::= \emptyset \mid x, \Gamma \mid \phi, \Gamma$
Resource contexts	$\Delta ::= \emptyset \mid A, \Delta$

The unrestricted context Γ contains universally quantified variables (parameters) x and pure propositions ϕ, which are duplicable. The resource context Δ contains Lithium *atoms* A. RefinedC's type assignments $\ell \lhd_l \tau$ and $v \lhd_\mathsf{v} \tau$ are atoms, but also user-defined propositions like `initialized` from Figure 4. As discussed in §4, type assignments and, thus, atoms are generally not duplicable.

Next, we describe goals, G. The simplest goals are *basic goals*, denoted F. Basic goals contain all RefinedC judgments such as $\vdash^{\Sigma}_{\mathsf{STMT}} s$ and $A_1 <: A_2\ \{G\}$. Building on these, goals 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 $-\!\!*$. We explain the exact purpose of this restriction later but, very briefly, it significantly narrows the search space.

***Goal-directed search in* Lithium.** The search for a proof of $\Gamma, \Delta \Vdash G$ is directed entirely 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) $G = \mathsf{True}$: Search succeeds trivially.
(2) $G = G_1 \wedge G_2$: Fork into two searches, one each for $\Gamma, \Delta \Vdash G_1$ and $\Gamma, \Delta \Vdash G_2$.
(3) $G = \forall x.\, G'(x)$: Introduce a new parameter y into Γ, and continue with the goal $G'(y)$.
(4) $G = \exists x.\, G'(x)$: Create a new evar $?x$ for x, and continue with the goal $G'(?x)$.
(5) $G = F$ (basic goal): A basic goal F represents a RefinedC judgment, so we look for a RefinedC rule $\frac{G'}{F'}$ whose conclusion F' matches F, and continue with the premise G' as the new goal.[21]
(6) $G = H * G'$: Here, we case analyze H:
   (a) $H = H_1 * H_2$: Replace the goal $(H_1 * H_2) * G'$ with the equivalent goal $H_1 * (H_2 * G')$. The next step will analyze the smaller formula H_1.
   (b) $H = \exists x.\, H'(x)$: Using the fact that the goal $(\exists x.\, H'(x)) * G'$ is equivalent to $\exists x.\, (H'(x) * G')$, create a new evar $?x$ for x, and continue with the smaller goal $H'(?x) * G'$.
   (c) $H = \ulcorner \phi \urcorner$: Generate the side condition ϕ in context Γ.
   (d) $H = A$: Find a hypothesis $A' \in \Delta$ that is *related to*[22] A, remove A' from Δ and continue with $A' <: A\,\{G'\}$.
(7) $G = H -\!\!* G'$: Here, we case analyze H:
   (a) $H = H_1 * H_2$: Replace goal $(H_1 * H_2) -\!\!* G'$ with equivalent goal $H_1 -\!\!* (H_2 -\!\!* G')$. The next step will analyze the smaller formula H_1.
   (b) $H = \exists x.\, H'(x)$: Using the fact that the goal $(\exists x.\, H'(x)) -\!\!* G'$ is equivalent to $\forall x.\, (H'(x) -\!\!* G')$, add a new parameter y to Γ, and continue with the smaller goal $H'(y) -\!\!* G'$.
   (c) $H = \ulcorner \phi \urcorner$: Add ϕ to Γ, and continue with goal G'.
   (d) $H = A$: Add A to Δ, and continue with goal G'.

These rules should be self-explanatory. Note that case (5) ($G = F$) is where RefinedC's *extensibility* reflects in Lithium: A basic goal F, which represents a RefinedC judgment, is established using an extensible set of user-defined rules.

Why* Lithium *does not need backtracking. Many (substructural) logic programming languages like Lolli also use goal-directed search, but they often need to backtrack, i.e., try several *alternatives* one by one. Lithium does not need backtracking, which makes its search faster, more predictable, and allows for better error messages. In the following, we explain the reasons for which standard logic programming needs backtracking and why these reasons do not apply to Lithium.

First, in standard logic programming, a goal of the form $G_1 * G_2$ is handled by *splitting* the resource context Δ into Δ_1, Δ_2 and then proving the two subgoals G_1 and G_2 in these contexts. This can cause *backtracking* as there may be several possible splits.[23] In Lithium, we rely on the fact that the left-subgoal G_1 must have a restricted form (H). This restricted form can be case-analyzed all the way down to atoms (case (6) and its subcases), which eliminates this source of backtracking.

A second source of backtracking in standard logic programming is the selection of hypothesis from Δ on which *elimination rules* can be applied to prove an atomic goal (so-called "backchaining"). In Lithium, Δ is limited to atoms, so elimination rules simply do not apply! Importantly, we are

[21] Technically, this involves unification: We unify F' and F, and apply the unifier to the new goal G'.
[22] Lithium is generic over the notion of relatedness of atoms, but RefinedC ensures that two type assignments for the same location (or value) are related. This encodes the meta-level rules <:-E-VAL and <:-E-LOC mentioned in §4.5.
[23] Backtracking may be needed in this case even with optimizations such as the input-output method of Lolli.

Class	Test	Impl	Spec	Annot	Pure	Imp	Overhead
#1	Singly linked list	98	28	23	0	0	~0.2
	Queue	42	15	12	0	0	~0.3
	Binary search	17	5	5	21	0	~1.5
#2	Thread-safe allocator	71	18	21	0	0	~0.3
	Page allocator	43	15	16	0	0	~0.4
#3	Binary search tree (layered)	133	65	27	122	0	~1.1
	Binary search tree (direct)	115	42	27	0	0	~0.2
#4	Hashmap with linear probing	111	46	57	240	0	~2.7
#5	Hafnium's mpool allocator	192	54	60	0	0	~0.3
#6	Spinlock	25	12	3	0	301	~12.2

Table 2. Evaluation. Impl: lines of code (without comments, counted by tokei [The Tokei Team 2020]). Spec: lines of function specification. Annot: lines of annotations in source code (*e.g.*, loop and data-structure invariants). Pure: lines of pure Coq reasoning. Imp: lines of impure type system and Iris reasoning (Pure and Imp including definitions and lemma statements). Overhead: Sum of Annot, Pure and Imp divided by Impl.

able to restrict Δ to atoms because, in goals of the form $H \mathrel{-\!\!*} G$, H is restricted. This again allows us to case analyze H down to atoms before introducing anything into Δ (case (7) and its subcases).

Nonetheless, in case (6d) there could be multiple hypothesis A' that are related to A in the goal. Here, we rely on the fact that only type assignments for the same location or value are related and, for each location or value, there is at most one assignment assumption at a time. Consequently, there is no ambiguity in matching atomic goals with hypotheses.

Finally, standard logic programming backtracks when it encounters disjunction in a goal. Here, we do not allow disjunction at all because RefinedC's rules do not require it.

In principle, in case (5) there could be multiple typing rules that match the basic goal. However, this is not a problem in practice as RefinedC typing rules usually do not overlap.[24]

Implementation. We have implemented a Lithium interpreter using the Ltac language [Delahaye 2000] of Coq. The search for matching RefinedC typing rules (case (5) above) is handled by Coq's typeclass mechanism [Sozeau and Oury 2008]. The interpreter maps Γ to the standard Coq context and Δ to the spatial context provided by the Iris Proof Mode [Krebbers et al. 2017b, 2018].

6 EVALUATION AND CASE STUDIES

In this section we report on our evaluation of RefinedC against closely related verification tools, and briefly discuss a selection of case studies highlighting the expressiveness of RefinedC. We consider six classes of examples for which we have verified *full functional correctness* in RefinedC. For each example the overhead in terms of annotations is summarized in Table 2.

#1: Comparison to similar tools. The first three examples of Table 2 refer to common case studies for verification tools. They are similarly covered by the four state-of-the-art verification

[24]Our Lithium implementation resolves the rare cases of overlapping rules via a simple priority system and does not backtrack on the choice of rules.

tools that we are comparing against: Bedrock [Chlipala 2011] and VST [Cao et al. 2018], which are both foundational, and VeriFast [Jacobs et al. 2011] and VCC [Cohen et al. 2009], which are not.

As Table 2 shows, the annotation overhead of RefinedC is low (mostly loop-invariants). Only the verification of binary search requires domain-specific reasoning (about sorted lists) not covered by RefinedC's automation. However, all ownership reasoning is automatically discharged, and only pure side conditions are left to the user.

In comparison, VCC has a large annotation overhead when verifying code with dynamic ownership changes (*e.g.*, insertion/deletion in a linked list). This is no surprise because VCC, unlike the other three tools, is not based on separation logic. On the other hand, VCC can verify binary search (which does not involve ownership manipulation) without additional annotations.

VeriFast's annotation overhead is similar to RefinedC's, thanks to its folding/unfolding heuristics for abstract predicates [Vogels et al. 2011]. RefinedC benefits from existing proof libraries and tactics for Coq, *e.g.*, [The Coq-std++ Team 2020], but no such reuse is directly possible for VeriFast.

On the foundational side, Bedrock provides mostly-automatic verification for an assembly-like language. Its proof automation is able to discharge most of the obligations arising from the verification. However, while RefinedC provides reusable abstractions via its type system, in Bedrock new predicates are built from the primitives of separation logic. Thus, the user has to provide folding and unfolding lemmas to guide the automation, while they come for free with RefinedC.

Among the four systems in our comparison, VST is the closest to RefinedC as both do foundational verification for a realistic model of C. While VST comes with some proof automation [Cao et al. 2018], it still requires manual proofs in Coq. Overall, this leads to a significantly higher overhead relative to RefinedC.

As an aside, the `append` function verified by RefinedC takes the address of the `next` field, similar to the code in Figure 5. All other tools use a (less idiomatic) version of `append` that makes an additional check for list emptiness at the beginning of the function to avoid taking the address.

#2: Ownership reasoning. To evaluate ownership reasoning, we verify several allocators. Here the RefinedC type system shows its expressiveness as all necessary ownership transformations (*e.g.*, O-ADD-UNINIT and similar rules for $\mathsf{padded}(\tau, n)$) are provided as typing rules.

#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 [Gu et al. 2018; Lorch et al. 2020]. To investigate how a layered approach works in RefinedC, we have verified a binary search tree first via an intermediate functional layer, and second by directly going from C to the desired specification of a functional set. While both approaches are viable with RefinedC, the overhead of the direct approach is significantly smaller as it does not require defining the intermediate layer. The direct approach works well because the type system cleanly separates the ownership reasoning from pure functional reasoning. In comparison, the layered approach is favored by VST, where ownership reasoning and pure functional reasoning is handled in the same proof.

#4: Tricky functional correctness reasoning. We illustrate RefinedC's ability to verify data structures with tricky invariants by verifying 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. The RefinedC type system reduces verification to pure reasoning about this invariant, which in turn is discharged via manual proofs in Coq.

#5: Verifying complex real-world code. The largest case study applies RefinedC to a version of the page allocator[25] of the Hafnium hypervisor [Hafnium 2020]. This verification combines many of the previously mentioned techniques, and shows that RefinedC can verify real-world C code.

#6: Building concurrent abstractions. We evaluate the ability of the RefinedC type system to incorporate new abstractions based on the spinlock abstraction used in §2.2. As Table 2 shows, verification of concurrent abstractions has a significantly higher overhead than the other examples, and perhaps more importantly requires impure reasoning using Iris. The benefit, however, is that no such impure reasoning is required for examples that *use* the abstraction (here, #2 and #5). Also, while the RefinedC type system cannot handle atomic instructions, it can verify the surrounding sequential code. This is possible because Lithium does not backtrack, and thus manual reasoning steps can be interleaved with the automatic type checker.

7 OTHER RELATED WORK

In §6, we compared RefinedC to both foundational and non-foundational tools for verification of low-level code. This section discusses a wider range of related work for both Lithium and RefinedC.

Logic programming languages for linear or separation logic. Previous work on logic programming languages for linear or separation logic [Andreoli 1992; Hodas and Miller 1994; Harland et al. 1996; Armelín and Pym 2001] focuses on identifying large subsets of the underlying logic that remain amenable to logic programming. However, this requires using expensive techniques like backtracking. In contrast, Lithium is deliberately limited to the smallest subset of separation logic that suffices for RefinedC. This subset does not require backtracking, and it is easier to implement a certifying interpreter for this subset.

Automation for separation logic. The literature abounds in automatic solvers for separation logic and frame inference [Piskac et al. 2014; Lee and Park 2014; Reynolds et al. 2016; Le et al. 2018; Ta et al. 2018]. These solvers assume more knowledge about the atomic formulas they operate on (usually a variant of the symbolic heap fragment [Berdine et al. 2004] of separation logic)—and they rely on more sophisticated automation (*e.g.*, based on SMT solvers)—than Lithium does. In contrast, proof search in Lithium is conceptually more straightforward (which makes it more predictable), but also extensible to new atoms and rules (as Lithium does not require atoms to have a particular shape). This extensibility is crucial for supporting custom abstractions like the spinlock shown in §2 and for covering the many reasoning patterns used by idiomatic C code.

Memory safety in low-level programming languages. A significant amount of prior work is focused on proving only *safety* (*e.g.*, memory safety) of low-level code. This is quite different from, and much simpler than, RefinedC's goal of full functional verification. However, since some of the techniques used are similar, we discuss this line of work as well.

Necula et al. [2002]; Condit et al. [2007]; Elliott et al. [2018] use a combination of *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 [Rondon et al. 2010] verify memory safety of C code using a combination of refinement types [Rondon et al. 2008] and alias types [Walker and Morrisett 2001]. Since their focus is on verifying the safety of array accesses and pointer dereferences, the annotation overhead is low (*e.g.*, no loop invariants are required), but the verification may be imprecise. In contrast, RefinedC targets full functional verification and requires more annotations, but it can also verify more programs (*e.g.*, it addresses the limitations described by Rondon et al. [2010, Section 5.1]). Finally, one approach to safety in low-level code is to use a

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

memory-safe language such as Vault [DeLine and Fähndrich 2001], Cyclone [Fluet et al. 2006], or Rust [The Rust Team 2020] 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 [Freeman and Pfenning 1991; Xi 2007], although originally developed for functional programs, have also been used for the safety and correctness of imperative code [Rondon et al. 2010; Bakst and Jhala 2015; Toman et al. 2020]. While this line of work usually focuses on fully automatic type systems for relatively simple imperative languages, RefinedC requires more annotations (*e.g.*, loop invariants), but can verify more complicated properties and supports a realistic model of C (*e.g.*, supporting uninitialized memory and pointer arithmetic). A closely related work in this area is ConSORT [Toman et al. 2020], which—similar to RefinedC—combines refinement types with ownership types. Both exploit a synergy between refinement types and ownership types: the latter help to support strong updates, which refinement types for imperative languages require. However, the concrete notions of refinement types and ownership types are quite different: in contrast to RefinedC's *indexed* refinement types (à la DML [Xi 2007]), ConSORT's *subset* refinement types (à la Liquid Types [Rondon et al. 2008]) disallow refinements that refer to memory locations. For example, ConSORT cannot express the type of a pair of pointers where both pointers point to the same value, while this is possible in RefinedC. ConSORT also has a fixed notion of ownership, whereas RefinedC's is extensible using a model based on Iris. This extensible notion enables us to define types in RefinedC like `spinlock<lid>`, which are not expressible using ConSORT's type system.

Prusti [Astrauskas et al. 2019] uses ownership types to automate separation logic-based verification. However, Prusti focuses only on safe Rust programs, so it does not support unsafe features like pointer arithmetic and uninitialized memory, which are common in C and supported by RefinedC.

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), and in part by the EPSRC Programme Grant REMS: Rigorous Engineering of Mainstream Systems (EP/K008528/1). Robbert Krebbers was supported by the Dutch Research Council (NWO), project 016.Veni.192.259.

REFERENCES

Jean-Marc Andreoli. 1992. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation* 2, 3 (1992), 297–347. https://doi.org/10.1093/logcom/2.3.297

Anonymous. 2020. RefinedC: An Extensible Refinement Type System for C Based on Separation Logic Programming (Artifact). Submitted as supplementary material (packaged with the appendix).

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

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *PACMPL* 3, OOPSLA (2019), 1–30. https://doi.org/10.1145/3360573

Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient detection of all pointer and array access errors. *PLDI* (1994). https://doi.org/10.1145/178243.178446

Alexander Bakst and Ranjit Jhala. 2015. Predicate abstraction for linked data structures. *VMCAI* (2015), 65–84. https://doi.org/10.1007/978-3-662-49122-5_3

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

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2004. A Decidable Fragment of Separation Logic. *LNCS* (2004), 97–109. https://doi.org/10.1007/978-3-540-30538-5_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. *JAR* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

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

Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. *PLDI* (2011). https://doi.org/10.1145/1993498.1993526

Adam Chlipala. 2015. From network interface to multithreaded web applications: A case study in modular program verification. *POPL* (2015). https://doi.org/10.1145/2676726.2677003

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. *TPHOLs* (2009), 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. 2008. Unifying type checking and property checking for low-level code. *POPL* (2008). https://doi.org/10.1145/1480881.1480921

Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. 2007. Dependent types for low-level programming. In *ESOP (LNCS, Vol. 4421)*. 520–535. https://doi.org/10.1007/978-3-540-71316-6_35

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

David Delahaye. 2000. A tactic language for the system Coq. In *LPAR (LNCS, Vol. 1955)*. 85–95. https://doi.org/10.1007/3-540-44404-1_7

Robert DeLine and Manuel Fähndrich. 2001. Enforcing high-level protocols in low-level software. *ACM SIGPLAN Notices* 36, 5 (2001), 59–69. https://doi.org/10.1145/381694.378811

Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C safe by extension. *SecDev* (2018). https://doi.org/10.1109/secdev.2018.00015

Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear regions are all you need. In *ESOP (LNCS, Vol. 3924)*. 7–21. https://doi.org/10.1007/11693024_2

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. *PLDI* (1991). https://doi.org/10.1145/113445.113468

Dan Frumin, Léon Gondelman, and Robbert Krebbers. 2019. Semi-automated reasoning about non-determinism in C expressions. In *ESOP (LNCS, Vol. 11423)*. 60–87. https://doi.org/10.1007/978-3-030-17184-1_3

David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. 2013. Don't sweat the small stuff: Formal verification of C code without the pain. *PLDI* (2013). https://doi.org/10.1145/2594291.2594296

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. *PLDI* (2018). https://doi.org/10.1145/3192366.3192381

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

James Harland, David Pym, and Michael Winikoff. 1996. Programming in Lygon: An overview. In *AMAST (LNCS, Vol. 1101)*. 391–405. https://doi.org/10.1007/bfb0014329

Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A framework for defining logics. *JACM* 40, 1 (1993), 143–184. https://doi.org/10.1145/138027.138060

Joshua S. Hodas and Dale Miller. 1994. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation* 110, 2 (1994), 327–365. https://doi.org/10.1006/inco.1994.1036

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

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), 1–34. https://doi.org/10.1145/3158154

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

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

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *POPL* (2015). https://doi.org/10.1145/2676726.2676980

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *ECOOP (LIPIcs, Vol. 74)*. 17:1–17:29. https://doi.org/10.4230/LIPIcs.

ECOOP.2017.17

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. In *SEFM (LNCS, Vol. 27)*. 573–609. https://doi.org/10.1007/s00165-014-0326-7

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

Robbert Krebbers. 2015. *The C standard formalized in Coq*. Ph.D. Dissertation. Radboud University Nijmegen.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The essence of higher-order concurrent separation logic. In *ESOP (LNCS, Vol. 10201)*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

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

Robbert Krebbers and Freek Wiedijk. 2013. Separation logic for non-local control flow and block scope variables. In *FoSSaCS (LNCS, Vol. 7794)*. 257–272. https://doi.org/10.1007/978-3-642-37075-5_17

Quang Loc Le, Jun Sun, and Shengchao Qin. 2018. Frame inference for inductive entailment proofs in separation logic. In *TACAS (LNCS, Vol. 10805)*. 41–60. https://doi.org/10.1007/978-3-319-89960-2_3

Wonyeol Lee and Sungwoo Park. 2014. A proof system for separation logic with magic wand. *POPL* (2014). https://doi.org/10.1145/2535838.2535871

Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert memory model, version 2*. Technical Report RR-7987. Inria.

Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

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*. 197–210. https://doi.org/10.1145/3385412.3385971

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. *PACMPL* 3, POPL (2019), 67:1–67:32. https://doi.org/10.1145/3290380

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. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016* (2016). https://doi.org/10.1145/2908080.2908081

George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. *POPL* (2002), 128–139. https://doi.org/10.1145/503272.503286

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *CSL (LNCS, Vol. 2142)*. 1–19. https://doi.org/10.1007/3-540-44802-0_1

Frank Pfenning and Carsten Schürmann. 1999. System description: Twelf - A meta-logical framework for deductive systems. In *CADE (LNCS, Vol. 1632)*. 202–206. https://doi.org/10.1007/3-540-48660-7_14

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. Automating separation logic with trees and data. In *CAV (LNCS, Vol. 8559)*. 711–728. https://doi.org/10.1007/978-3-319-08867-9_47

François Pottier and Jonathan Protzenko. 2013. Programming with permissions in Mezzo. *ICFP* (2013). https://doi.org/10.1145/2500365.2500598

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

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. (2002), 55–74. https://doi.org/10.1109/LICS.2002.1029817

Patrick Maxim Rondon. 2012. *Liquid Types*. Ph.D. Dissertation. University of California, San Diego, USA. http://www.escholarship.org/uc/item/1646v8mx

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

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. *PLDI* (2008). https://doi.org/10.1145/1375581.1375602

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

Matthieu Sozeau and Nicolas Oury. 2008. First-class type classes. In *TPHOLs (LNCS, Vol. 5170)*. 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

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

The Coq-std++ Team. 2020. An extended "standard library" for Coq. Available online at https://gitlab.mpi-sws.org/iris/stdpp.

The Coq Team. 2020. The Coq proof assistant. https://coq.inria.fr/.

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

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

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)*. 684–714. https://doi.org/10.1007/978-3-030-44914-8_25

Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans. 2011. Annotation inference for separation logic based verifiers. In *Formal Techniques for Distributed Systems (LNCS, Vol. 6722)*. 319–333. https://doi.org/10.1007/978-3-642-21461-5_21

David Walker and Greg Morrisett. 2001. Alias types for recursive data structures. *Types in Compilation* (2001), 177–206. https://doi.org/10.1007/3-540-45332-6_7

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

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

A TYPING RULES FOR $\vdash^{\Sigma}_{\text{STMT}}$ AND $\vdash_{\text{EXPR}}$

This section presents the fixed typing rules for $\vdash^{\Sigma}_{\text{STMT}}$ (in §A.1) and $\vdash_{\text{EXPR}}$ (in §A.2). There are too many typing rules for the specialized judgments to present them in this appendix. The interested reader is referred to the accompanying Coq development.

A.1 Typing rules for $\vdash^{\Sigma}_{\text{STMT}}$

The goto b statement requires special treatment by the RefinedC type system as it has two different typing rules depending on whether the loop is annotated with a loop invariant H (represented by the atomic assertion $\text{b} \triangleleft^{\Sigma}_{\text{BLOCK}} H$) or not. Thus, the implementation of the type checker special cases goto b to apply the correct rule. This is the only statement for which such a special case is required.

T-GOTO-PRECOND
$$\frac{\exists H.\ (\text{b} \triangleleft^{\Sigma}_{\text{BLOCK}} H) * H * \text{True}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{goto}\ \text{b}}$$

T-GOTO
$$\frac{\vdash^{\Sigma}_{\text{STMT}} C[\text{b}]}{\vdash^{\Sigma}_{\text{STMT}} \texttt{goto}\ \text{b}}\ \Sigma = (C,\ \overline{(\ell, n)},\ \exists x.\ \tau(x); H(x))$$

T-ASSIGN
$$\frac{\vdash_{\text{EXPR}} e_2\ \{v_2, \tau_2. \exists \tau.\ \ell_1 \triangleleft_l \tau * \vdash_{\text{PLACE}} K[\ell_1 : \tau]\ \{\ell_3, \tau_3, T.\ \vdash_{\text{WRITE}} \ell_3 : \tau_3 \leftarrow v_2 : \tau_2\ \{\tau_4. \ell_1 \triangleleft_l T[\tau_4] \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}} s\}\}\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{store}(p_1, e_2);\ s}\ p_1 = K[\ell_1]$$

T-RETURN
$$\frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ \exists x.\ v \triangleleft_{\text{v}} \tau(x) * \overline{\ell \triangleleft_l \text{uninit}(n)} * H(x)\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{return}\ e}\ \Sigma = (C,\ \overline{(\ell, n)},\ \exists x.\ \tau(x); H(x))$$

T-IF
$$\frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ \vdash^{\Sigma}_{\text{IF}} v : \tau\ \texttt{then}\ s_1\ \texttt{else}\ s_2\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

T-SWITCH
$$\frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ \vdash^{\Sigma}_{\text{SWITCH}} \texttt{switch}_\iota v : \tau\ \texttt{case}\ \overline{s_1}\ \texttt{default}\ s_2\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{switch}_\iota\ e\ \texttt{case}\ \overline{s_1}\ \texttt{default}\ s_2}$$

T-CALL
$$\frac{\vdash_{\text{EXPR}} e_f\ \{v, \tau.\ \exists \overline{\tau_{\text{arg}}}.\ \exists H_1.\ \exists \tau_{\text{ret}}.\ \exists H_2.\ v \triangleleft_{\text{v}} (\text{fn}(\forall x.\ \overline{\tau_{\text{arg}}(x)}; H_1(x)) \to \exists y.\ \tau_{\text{ret}}(x, y); H_2(x, y)) * \exists x.\ \vdash_{\text{EXPR}} e\ \{v', \tau'.\ v' \triangleleft_{\text{v}} \tau_{\text{arg}}(x) * H_1(x) \mathrel{-\!\!*} \forall v_{\text{ret}}.\ \forall y.\ v_{\text{ret}} \triangleleft_{\text{v}} \tau_{\text{ret}}(x, y) \mathrel{-\!\!*} H_2(x, y) \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}} s[\text{i} \mapsto v_{\text{ret}}]\}\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{let}\ \text{i} = \texttt{call}\ e_f(\overline{e});\ s}$$

T-ASSERT
$$\frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ \vdash^{\Sigma}_{\text{ASSERT}} v : \tau\ ; s\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{assert}(e);\ s}$$

T-ANNOTS
$$\frac{\exists \tau.\ \ell \triangleleft_l \tau * \vdash_{\text{PLACE}} K[\ell : \tau]\ \{\ell_2, \tau_2, T.\ \vdash_{\text{ADDR}} \ell_2 : \tau_2\ \{\tau_3, \tau_2'.\ \ell \triangleleft_l T[\tau_2'] \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{ANNOTSTMT}} \texttt{annot}_x\ \ell_2 : \tau_2; s\}\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{annot}_x \& p;\ s}\ p = K[\ell]$$

T-EXPRS
$$\frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ v \triangleleft_{\text{v}} \tau \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}} s\}}{\vdash^{\Sigma}_{\text{STMT}} e;\ s}$$

T-SKIPS
$$\frac{\vdash^{\Sigma}_{\text{STMT}} s}{\vdash^{\Sigma}_{\text{STMT}} \texttt{skip};\ s}$$

A.2 Typing rules for $\vdash_{\text{EXPR}}$

$$\text{T-VAL}\quad \frac{\vdash_{\text{VAL}} v \xrightarrow{\tau} G(v,\tau)}{\vdash_{\text{EXPR}} v \{v', \tau.\ G(v',\tau)\}} \qquad \text{T-UNOP}\quad \frac{\vdash_{\text{EXPR}} e \{v, \tau.\ \vdash_{\text{UNOP}} \odot v : \tau \{v_2, \tau_2.\ G(v_2,\tau_2)\}\}}{\vdash_{\text{EXPR}} \odot e \{v, \tau.\ G(v,\tau)\}}$$

$$\text{T-BINOP}\quad \frac{\vdash_{\text{EXPR}} e_1 \{v_1, \tau_1.\ \vdash_{\text{EXPR}} e_2 \{v_2, \tau_2.\ \vdash_{\text{BINOP}} v_1 : \tau_1 \odot v_2 : \tau_2 \{v_3, \tau_3.\ G(v_3,\tau_3)\}\}\}}{\vdash_{\text{EXPR}} e_1 \odot e_2 \{v, \tau.\ G(v,\tau)\}}$$

$$\text{T-SKIPE}\quad \frac{\vdash_{\text{EXPR}} e \{v, \tau.\ G(v,\tau)\}}{\vdash_{\text{EXPR}} \texttt{skip}; e \{v, \tau.\ G(v,\tau)\}}$$

$$\text{T-USE}\quad \frac{\exists \tau.\ \ell \lhd_l \tau * \vdash_{\text{PLACE}} K[\ell : \tau] \{\ell_2, \tau_2, T.\ \vdash_{\text{READ}} \tau_2 \{v_3, \tau_2', \tau_3.\ \ell \lhd_l T[\tau_2'] \mathrel{-\!\!*} G(v_3,\tau_3)\}\}}{\vdash_{\text{EXPR}} \texttt{use}(p) \{v, \tau.\ G(v,\tau)\}}\ p = K[\ell]$$

$$\text{T-ADDR-OF}\quad \frac{\exists \tau.\ \ell \lhd_l \tau * \vdash_{\text{PLACE}} K[\ell : \tau] \{\ell_2, \tau_2, T.\ \vdash_{\text{ADDR}} \ell_2 : \tau_2 \{\tau_3, \tau_2'.\ \ell \lhd_l T[\tau_2'] \mathrel{-\!\!*} G(\ell_2, \&_{\text{own}}(\tau))\}\}}{\vdash_{\text{EXPR}} \&p \{v, \tau.\ G(v,\tau)\}}\ p = K[\ell]$$

$$\text{T-ANNOTE}\quad \frac{\vdash_{\text{EXPR}} e \{v, \tau.\ \vdash_{\text{ANNOTEXPR}} \texttt{annot}_x\ v : \tau \{G(v,\tau)\}\}}{\vdash_{\text{EXPR}} \texttt{annot}_x(e) \{v, \tau.\ G(v,\tau)\}}$$

B SUMMARY OF THE JUDGMENTS OF REFINEDC

Table 3 shows the typing judgments used by RefinedC and gives their semantic interpretation (except for the judgments for l-expressions, which are described informally). The typing judgment for expressions $\vdash_{\text{EXPR}}$ is defined using the standard weakest precondition provided by Iris [Jung et al. 2018b]. The typing judgment for statements $\vdash^{\Sigma}_{\text{STMT}}$ uses the weakest precondition for Caesium statements $\text{wp}^C\ s\ \{\Phi\}$, which is parametrized by the control-flow graph C and derived from the standard Iris weakest precondition.

class	judgment	description / semantic equivalent
statements	$\vdash^{(C,\overline{(\ell,n)},\exists x.\ \tau(x);H(x))}_{\text{STMT}}\ s$	$\text{wp}^C\ s\left\{v.\exists x.\ v \lhd_{\text{v}} \tau(x) * \overline{\ell \lhd_{\text{l}} \text{uninit}(n)} * H(x)\right\}$
	$\vdash^{\Sigma}_{\text{IF}}\ v : \tau\ \texttt{then}\ s_1\ \texttt{else}\ s_2$	$v \lhd_{\text{v}} \tau \twoheadrightarrow \vdash^{\Sigma}_{\text{STMT}}\ \texttt{if}\ v\ \texttt{then}\ s_1\ \texttt{else}\ s_2$
	$\vdash^{\Sigma}_{\text{SWITCH}}\ \texttt{switch}_{\alpha} v : \tau\ \texttt{case}\ \overline{s}\ \texttt{default}\ s'$	$v \lhd_{\text{v}} \tau \twoheadrightarrow \vdash^{\Sigma}_{\text{STMT}}\ \texttt{switch}_{\alpha}\ v\ \texttt{case}\ \overline{s}\ \texttt{default}\ s'$
	$\vdash^{\Sigma}_{\text{ASSERT}}\ v : \tau\ ; s$	$v \lhd_{\text{v}} \tau \twoheadrightarrow \vdash^{\Sigma}_{\text{STMT}}\ \texttt{assert}(v);\ s$
	$\vdash^{\Sigma}_{\text{ANNOTSTMT}}\ \texttt{annot}_{x}\ \ell : \tau; s$	$\ell \lhd_{\text{l}} \tau \twoheadrightarrow \vdash^{\Sigma}_{\text{STMT}}\ s$
r-expressions	$\vdash_{\text{EXPR}}\ e\ \{v, \tau.\ G(v,\tau)\}$	$\text{wp}\ e\ \{v.\exists \tau.\ v \lhd_{\text{v}} \tau * G(v,\tau)\}$
	$\vdash_{\text{BINOP}}\ v_1 : \tau_1 \odot v_2 : \tau_2\ \{v, \tau.\ G(v,\tau)\}$	$v_1 \lhd_{\text{v}} \tau_1 \twoheadrightarrow v_2 \lhd_{\text{v}} \tau_2 \twoheadrightarrow \vdash_{\text{EXPR}}\ v_1 \odot v_2\ \{v, \tau.\ G(v,\tau)\}$
	$\vdash_{\text{UNOP}}\ \odot v : \tau\ \{v', \tau'.\ G(v',\tau')\}$	$v \lhd_{\text{v}} \tau \twoheadrightarrow \vdash_{\text{EXPR}}\ \odot v\ \{v', \tau'.\ G(v',\tau')\}$
	$\vdash_{\text{VAL}}\ v \xrightarrow{\tau} G(\tau)$	$\exists \tau.\ v \lhd_{\text{v}} \tau * G(v,\tau)$
	$\vdash_{\text{ANNOTEXPR}}\ \texttt{annot}_{x}\ v : \tau\ \{G\}$	$v \lhd_{\text{v}} \tau \twoheadrightarrow G$
l-expressions	$\vdash_{\text{PLACE}}\ K[\ell : \tau]\ \{\ell_2, \tau_2, T.\ G(\ell_2, \tau_2, T)\}$	accessing ℓ with type τ using evaluation context K resulting in ℓ_2 with type τ_2 and ℓ having type T with hole
	$\vdash_{\text{READ}}\ \tau\ \{v_2, \tau', \tau_2.\ G(v_2, \tau', \tau_2)\}$	reading from a location with type τ resulting in v_2 with type τ_2 and the location having type τ'
	$\vdash_{\text{WRITE}}\ \ell_1 : \tau_1 \leftarrow v_2 : \tau_2\ \{\tau_3.\ G(\tau_3)\}$	writing v_2 with type τ_2 to ℓ_1 with type τ_1 resulting in ℓ_1 having type τ_3
	$\vdash_{\text{ADDR}}\ \ell : \tau\ \{\tau_2, \tau'.\ G(\tau_2, \tau')\}$	taking address of ℓ with type τ resulting in τ_2 and ℓ having type τ'
Auxiliary judgments	$A_1 <: A_2\ \{G\}$	$A_1 \twoheadrightarrow A_2 * G$

Table 3. Judgments