Fulminate: Testing CN Separation-Logic Specifications in C

RINI BANERJEE, KAYVAN MEMARIAN, DHRUV MAKWANA, CHRISTOPHER PULTE, NEEL KRISHNASWAMI, and PETER SEWELL, University of Cambridge, UK

Separation logic has become an important tool for formally capturing and reasoning about the ownership patterns of imperative programs, originally for paper proof, and now the foundation for industrial static analyses and multiple proof tools. However, there has been very little work on program *testing* of separation-logic specifications in concrete execution. At first sight, separation-logic formulas are hard to evaluate in reasonable time, with their implicit quantification over heap splittings, and other explicit existentials.

In this paper we observe that a restricted fragment of separation logic, adopted in the CN proof tool to enable predictable proof automation, also has a natural and readable *computational* interpretation, that makes it practically usable in runtime testing. We discuss various design issues and develop this as a C+CN source to C source translation, Fulminate. This adds checks – including ownership checks and ownership transfer – for C code annotated with CN pre- and post-conditions; we demonstrate this on nontrivial examples, including the allocator from a production hypervisor. We formalise our runtime ownership testing scheme, showing (and proving) how its reified ghost state correctly captures ownership passing, in a semantics for a small C-like language.

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

Additional Key Words and Phrases: C, verification, separation logic, refinement types, runtime testing, pKVM, Android

ACM Reference Format:
Rini Banerjee, Kayvan Memarian, Dhruv Makwana, Christopher Pulte, Neel Krishnaswami, and Peter Sewell. 2025. Fulminate: Testing CN Separation-Logic Specifications in C. *Proc. ACM Program. Lang.* 9, POPL, Article 43 (January 2025), 33 pages. https://doi.org/10.1145/3704879

1 Introduction

Assurance of systems software has been a critical problem for decades, and is increasingly so in the modern threat environment, but it remains practically out of reach for the production systems code that widely-used computer systems rely on. There are many reasons for this – among the most important is the fact that such code is largely written in C or C++, with the investment in this codebase making it impractical to rewrite it in bulk. These are imperative languages in which programmers manually manage memory with complex ownership patterns, which are notoriously hard even to describe, let alone to reason about.

Separation logic has become one of the most important tools for formally capturing and reasoning about the ownership patterns of imperative programs. Originally used for paper proof [43], it is used in industrial static analyses, notably Infer [1, 8], and for multiple proof tools, including Iris [22], VST [10, 56], VeriFast [18], CN [41], Viper [36], Prusti [4], Gillian [30, 46], FCSL [48], and others.

Authors' Contact Information: Rini Banerjee, rb2018@cam.ac.uk; Kayvan Memarian, Kayvan.Memarian@cl.cam.ac.uk; Dhruv Makwana, dcm41@cam.ac.uk; Christopher Pulte, cp526@cam.ac.uk; Neel Krishnaswami, nk480@cl.cam.ac.uk; Peter Sewell, Peter.Sewell@cl.cam.ac.uk, University of Cambridge, Cambridge, UK.

ACM 2475-1421/2025/1-ART43
https://doi.org/10.1145/3704879

However, there has been very little work on program *testing* using separation-logic specifications (we discuss some exceptions in §6). We see two main reasons for this.

First, there is the cultural reason that testing and verification have largely been the focus of different communities, one focussed on bug-finding and the other on high assurance. There is an important potential intersection: the ability to use the same specifications both for runtime testing and for verification offers potential benefits for both, e.g. for quickly discovering some code and specification errors before embarking on proof, or for runtime testing of remaining proof obligations. This idea dates back at least to the 1970s, e.g. in Euclid [24] and Anna [28], and can be seen in the modern runtime verification, contracts, and gradual typing communities – but it arguably remains under-appreciated in the field of verification as a whole. A prominent recent example is the Frama-C E-ACSL [15, 51, 52]. E-ACSL is an executable fragment of the ACSL specification language that can be compiled to in-line C code for runtime assertion checking, but it, like other specification languages of the time, is based on first-order logic (albeit extended with some built-in predicates for heap footprints), making it hard to express complex ownership patterns.

Second, there is the technical reason that separation-logic formulas appear at first sight to be very hard to evaluate in reasonable time, even on a concrete memory state. A separating conjunction

$$P * Q$$

holds for a heap h if there exists some partition of h into two heaps h_1 and h_2 with disjoint footprint such that P holds for h_1 and Q holds for h_2, and that existential quantification has size $2^{\mathrm{size}(h)}$. Separation logic formulas often also involve existentials over memory addresses or values, e.g. to say that there exists some successor node y in a linked list (with no payload) linked from a first node at x:

$$\exists y,\, z.\; x \mapsto y * y \mapsto z$$

The CN proof system of Pulte et al. [41] uses a separation-logic refinement type inference system, with an SMT backend, for C verification. CN aims for as predictable a verification user experience as possible, to either cleanly accept or reject each input program rather than timing out or reporting 'unknown' when heuristics or the SMT solver fail (with a fallback to interactive Rocq proof when this is not possible). To that end, it is based on a type inference system that generates SMT queries only in the decidable SMT fragment. It restricts its specification language to a novel syntax for separation logic, using variable scoping to ensure that programmers write formulas in a way that inference can always succeed – a syntactic analogue of the input/output mode checks used in earlier systems such as VeriFast [18] and Gillian [30, 46].

The key observation of the current work is that this simple restriction is almost exactly what one needs for separation-logic specifications, of C function preconditions, postconditions, etc., to also have a natural and readable *computational* interpretation, that could be used in runtime testing. Our contributions are:

- We identify that this restriction, introduced previously for proof, is also good in principle for specification executability (§2.1 and §2.4).
- We describe various design issues involved in making a practical system based on it, using reified ghost state to record ownership information at runtime (§2.2).
- We provide a simple formalisation of our runtime ownership testing scheme, showing how this reified ghost state correctly captures ownership passing, in a semantics for a small C-like language with explicit ownership passing (§3).
- We describe the actual system, Fulminate (to make false CN specifications fail fast, especially on ownership errors), and the host of additional issues and engineering required to make

this work for a substantial fragment of real C, as a C+CN to C source translation that inlines checks of CN specifications (§4).
- We demonstrate all this on nontrivial examples, including the CN tutorial examples, a subset of the VeriFast tutorial [19] examples, a simple hashtable from Verifiable C [3], and, much more substantially, a "buddy" allocator from a production hypervisor, Google's pKVM developed for and deployed to protect virtual machines and the Android kernel from each other (§5).

We conclude with discussion of related (§6) and future work (§7). Together with the earlier work on CN, this gives a separation-logic specification language for which the same specifications can be used both for runtime testing and for proof, that is reasonably expressive w.r.t. separation and ownership properties. The main challenges were the observation of the first bullet (simple and obvious in hindsight but not in the previous proof-focussed research on separation logic), setting up the formal model to realistically capture the main ideas without largely-irrelevant C complexity, and making this work for a substantial fragment of real C, which is considerably more involved than one might think at first.

2 Key Ideas and High-Level Design

2.1 The CN Specification Language Looks Executable in Principle

We begin by recalling the CN specification language, originally designed for predictable proof, and illustrate on two simple examples how it intuitively has a computational interpretation for testing. Consider the C function on the right, that takes a pointer to a memory location containing a pointer to an integer, and zeros the latter. In classic separation logic, a natural specification of this would be a precondition saying that, on entry, there exists some address r and integer v such that the memory at p contains r, the memory at r contains v, and we have ownership of both p and r, with a postcondition saying that the same ownership is returned, with p still containing some r but with r containing 0:

```
void f(int **p) {
  int *q = *p;
  *q = 0;
}
```

$$\{\exists r, v.\ p \mapsto r * r \mapsto v\}\ \texttt{f(p)}\ \{\exists r.\ p \mapsto r * r \mapsto 0\}$$

Naive runtime checking of this would require a search for r and v – but, at least for this example, the values of r and v on entry are obviously determined by the value of `p` and the heap.

The CN specification language captures this with simple variable scoping: instead of primitive $x \mapsto y$ points-to predicates, separating conjunction of arbitrary formulas, and existential quantification, it combines ownership assertions and binders. The CN formula:

```
take r = Owned(p); P
```

both asserts ownership of location `p`, and binds the value at that location to `r` in the remaining predicate `P`. It is equivalent to the classic separation logic $\exists r.\ p \mapsto r * P$, whereas quantification of the left-hand-side of a points-to, $\exists p.\ p \mapsto r * P$, is intentionally not always CN-expressible.

CN supports such formulas in function pre- and post-conditions embedded in C source, as on the right. For runtime checking of a `take r1 = Owned(p);P` in a **`requires`** precondition, one has to check that the caller provides ownership of the memory at `p`, read its value from the actual memory on function entry, and evaluate `P` with `r1` replaced by that value. For runtime checking of `take r2 = Owned(p);P` in an **`ensures`** postcondition, one has to check that the current function has ownership of the memory at `p`, read its value from the actual memory on function exit, and evaluate `P` with `r2` replaced by that

```
void f(int **p)
/*@ requires take r1 = Owned(p);
             take v1 = Owned(r1);
    ensures take r2 = Owned(p);
            take v2 = Owned(r2);
            r2==r1 && v2==0i32;@*/
{ int *q = *p;
  *q = 0; }
```

value. For any access in the body of the function, e.g. the `*p`, one has to check that this function has ownership of that memory.

Separation logic uses inductively defined predicates to describe pointer data structures. For example, C linked lists, say of `node`s that each contain an integer and a (possibly null) pointer to another node:

```
struct node { int x; struct node *next; };
```

can be captured by the predicate `IntList(`p`, `xs`)` on the left below, that asserts ownership of a linked list starting at pointer p and says that list can be viewed abstractly as the mathematical list xs.

```
IntList(p, xs) =
  ( p = NULL ∧
    xs = nil )
  ∨
  ( ∃hd, q, tl.
    (p ↦ (hd, q) ∗
     IntList(q, tl)) ∧
    xs = cons(hd, tl)
  )
```

```
predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

On the face of it, that is hard to execute on a concrete memory: the general disjunction might need backtracking, the existential quantification might need search for values, and the $*$ needs search for memory splittings.

Now contrast with the CN version of this predicate definition, on the right above. This CN resource predicate looks like a function definition, that takes a **`pointer`** argument and returns a mathematical `integer_list` result, while also asserting ownership of the list nodes. It uses a conditional **`if`**…**`else`** instead of disjunction to avoid backtracking. The two **`take`**s, of the built-in predicate **`Owned`** and the recursive call to `IntList`, assert ownership of the list node at `p` and the remaining linked list, respectively, and also bind the mathematical values returned by each, to identifiers `node` and `tl` in the continuations.

The simple but powerful observation underlying this paper is that such CN predicates can be straightforwardly evaluated, over concrete memory states augmented with some reified ghost state recording ownership. Executing the CN `IntList(p)` requires walking over the heap, reading values and asserting ownership for **`take`**s of the primitive **`Owned`** predicate, constructing the mathematical-list result, and recursing as needed.

We conjecture that this computational reading will be natural and understandable for non-verification-aware programmers, for whom fancier logical constructs (quantifiers and inductive predicates, and indeed mode analysis) may be quite foreign. User studies will be required to determine whether this is true in practice.

2.2 The Fulminate Design Goals for CN Specification Testing

The above observation is simple, especially in hindsight, but going from that to a practical system for runtime testing of CN assertions for actual C code requires one to consider many design and implementation questions. We begin with some high-level goals and how they have determined our design.

Runtime testing on concrete executions. Fulminate supports runtime testing of CN assertions, covering both their ownership and pure-value aspects, in concrete execution of the underlying C

program. Like other runtime testing approaches, it is thus testing whether pre- and post-conditions hold of the concrete states reached at those program points within a complete execution, not whether pre-conditions imply post-conditions.

Flexible usage models for a shallow on-ramp to higher assurance. Fulminate aims at a very light-weight user experience, providing a gentle on-ramp to higher assurance in a variety of testing and testing+proof workflows. Conventional developers, who are not verification experts, might start by writing partial CN specifications – initially not much more than C **`assert`**s – and get immediate benefits from them by quickly detecting errors in CN testing, without any proof whatsoever. They could then gradually add richer ownership and functional-correctness properties. In higher-assurance contexts, verification experts might extend these and write more specifications for proof, in exactly the same CN specification language, and use Fulminate to quickly debug both their specifications and their code by testing them, interleaved with using CN refinement-type proof to verify them.

Testing of (a substantial fragment of) actual C. Fulminate supports a substantial fragment of actual C, rather than an idealised C-like language (though there are a number of features that are not yet supported, as we describe in §7). To this end, it uses the front-end of the Cerberus C semantics by Memarian et al. [32–34] to parse, typecheck, and desugar C source files, as does CN. To inject the translations of CN specifications into the C code, we have developed new machinery as a part of Cerberus.

Testing using conventional build processes and tools, and with arbitrary existing test harnesses and tests. Like systems from Anna [28] to the Frama-C E-ACSL [15, 51, 52], Fulminate is structured as an annotated-C source to C source translation, which adds instrumentation to check the CN specifications. The result of this translation can be built with a conventional C compiler, just like the original C source. The instrumented binary can then be executed in the same way as the original program, in whatever test harness and on whatever test suite the developers normally use. It is an on-line checker, obviating the need for logging that an off-line checker would have. Fulminate checking (optionally) involves checking every memory access, so logs would quickly become very large.

Design for usability: error reporting. Design for usability also makes it essential to give good error reports on runtime test failures, including e.g. both the C source location and CN predicate-definition source location on runtime ownership and other failures. We plan also to include optional ghost state to support reporting of which CN predicates own which parts of the heap, and thus which are involved in ownership errors. Even highly expert kernel developers may not have the ownership patterns of their code explicitly in mind, as they currently have no way to write them down, beyond prose comments, and no way to test or verify them. CN specifications and such feedback from testing may help develop a solid intuition and understanding of this.

Design for usability: debugging dynamic failures. Design for debuggability leads us to build a translation that closely follows the structure of the source program, preserving its layout and comments, and with the injected instrumentation in human-readable C. This means that one can, for example, sensibly use gdb or lldb on the generated code – including examination of the abstract states computed by CN predicates, and the reasons why CN assertions fail.

Minimal dependencies for bare-metal execution. One of our intended targets is systems code that runs in restricted execution environments, including hypervisor code such as Google's Android pKVM hypervisor [14], which runs as an isolated component at Arm-A Exception Level (EL) 2, above the Android kernel at EL1 and user code at EL0. Such code has to work in a very limited context, without the C standard library or most operating-system services. Fulminate has not yet

been demonstrated running at EL2, but this goal leads us to make the generated instrumentation essentially bare-metal C, without runtime dependencies.

Performance. Runtime testing is intrinsically expensive compared with uninstrumented execution, and runtime testing of separation logic predicates and ownership has to walk the relevant part of the heap, and check and/or update the corresponding ghost state, whenever such a specification is reached. Compared with sanitisers, for example, Fulminate is checking much richer properties, and one should expect it to have substantially more performance cost. However, Fulminate targets relatively small bodies of critical C code. Our working hypothesis is that, for such code operating over relatively small datastructures, even a relatively naive implementation is fast enough to be useful, and to explore how it behaves in practice – so long as it is on-line and translates to reasonable C, as described above. There are many potential optimisations, and the underlying scheme would support sampling – checking only a random subset of function executions – but we do not explore them in this paper.

Test generation. Fulminate does not itself do test-case *generation*, but it should work well with property-based testing: if one can automatically synthesise unit test cases, benefitting from CN ownership specifications for function preconditions, then one will be able to test the rich CN properties in unit-test executions, making performance much less critical. Fulminate may also work well with fuzzing, guiding fuzzing with rich-specification failures.

2.3 Ghost State Instrumentation for Ownership

The CN proof tooling checks each function in isolation, symbolically, in the usual program-logic fashion: for each function `f`, it assumes its **`requires`** `P` precondition and checks that implies that the body is free of undefined behaviour, and that the body establishes the **`ensures`** `Q` postcondition. For a call to another function `g` within the body of `f`, this checking uses the specification of `g` rather than its implementation: it checks that `P'` holds at that program point and assumes `Q'` afterwards. As it is a resource-aware logic, this involves checking that the resources of `P'` are available at that point, and it returns the resources of `Q'` to the remaining verification of `f`.

```
ty f(...parameters...)
/*@ requires P
    ensures  Q  @*/
{
  ...
  g(...);
  ...
  return ...;
}
```

```
ty' g(...parameters...)
/*@ requires P'
    ensures  Q'  @*/
{
  ...
  return ...;
}
```

Runtime testing is instead based on whole-program execution. For runtime testing of ownership properties, one should think in terms of *explicit resource passing*: at the call to `g`, the `P'` resources that `g` requires have to be taken from the ambient available resources (flagging a dynamic failure if that is not possible), and on return from `g`, the `Q'` resources that `g` ensures have to be returned. Pre- and post-conditions can also constrain pure properties of the values, as in the **`take`** `v2=`**`Owned`**`(r2); r2==r1 && v2==0i32` above, which also need to be checked and flag dynamic failures if false.

Resources are created for global variables, function parameters (which in C are mutable variables within the function body), block-scoped local variables, and allocators, and destroyed at function

return, block-end, and `free()`. For each memory access, runtime testing should check that its footprint is in the ambient available resources, flagging a dynamic failure otherwise.

In Section 3 we make this precise with an ownership-passing operational semantics for a small C-like calculus, that includes globals, C-like function parameters, and block-scoped locals, and a small fragment of the CN specification language.

For the actual Fulminate implementation, however, it is more convenient to use a rather different but essentially isomorphic representation. Instead of passing the ambient resources around explicitly, we maintain reified ghost state that encodes a finite partial function that, for each byte of memory, gives the C stack depth of the C function holding ownership of that byte (if any). This gives a simple implementation scheme, in outline:

(1) At each ownership creation point (globals, function parameters, block-scope entry) we add instrumentation to map the footprint of the created variable to the current stack depth.
(2) At the corresponding ownership destruction points (function exit and block-scope exit) we add instrumentation to unmap the appropriate footprint.
(3) For each **`requires`** `P` clause, we add instrumentation that computes the CN predicate `P`, walking over the heap as needed, checking that the required resources are mapped at the immediate caller's stack depth, and updating them to the current stack depth.
(4) For each **`ensures`** `P` clause, we add instrumentation that computes the CN predicate `P` in a different mode: walking over the heap as needed, checking the required resources are mapped at the current stack depth, and updating them to the caller's stack depth. We also optionally check that no resources are leaked.
(5) For each memory access, we add instrumentation that checks that the footprint of the access is owned at the current stack depth.

This supports, for example, passing the address of a local variable, which might transiently be involved in some pointer datastructure spanning it and parts of the heap, down to a callee function. In Section 3 we also make this precise, with an instrumented version of the operational semantics, and prove the two coincide, and in Section 4 we describe our implementation, which has to handle many more issues.

2.4 The CN Specification Language

The CN specification language, that Fulminate translates into runtime checks, is a first-order language that syntactically distinguishes linear resource types and freely duplicable constraint types, featuring (possibly mutually recursive) definitions of resource predicates, specification functions, and datatypes. To explain what Fulminate has to translate, we briefly describe here (almost) the full CN specification language.

C functions are specified using **`requires`** and **`ensures`** *condition* lists, where each condition is either a *resource* **`take`** binding, a *constraint*, or a let binding. Resources comprise the built-in predicates **`Owned`** (points-to indexed by C-types) and **`Block`** (points-to for uninitialised memory), as well as user-defined predicates and iterated separating conjunctions of predicates: an **`each`**$(bty\ x;\ t)\{p(t_1,\ldots,t_n)\}$ asserts ownership of resource $p(t_1,\ldots,t_n)$ for each instance of the quantified x, of type bty, for which the guard expression t holds. Constraints are boolean-typed expressions, possibly universally quantified using **`each`**.

toplevel_def ::= *c_function_def* | *spec_predicate_def* | *spec_function_def* | *datatype_def*
c_function_def ::= $cty\ f\ (cty_1\ id_1, .., cty_n\ id_n)$ /*@ **`requires`** *conditions* **`ensures`** *conditions′* @*/ *block*
condition ::= **`take`** *id* = *resource* | *constraint* | **`let`** *id* = *t* *conditions* ::= $condition_1; \ldots; condition_n;$
resource ::= $p(t_1, .., t_n)$ | **`each`** $(bty\ id; t)\ \{\ p(t_1, .., t_n)\}$
pred_name, *p* ::= **`Owned`**<*cty*> | **`Block`**<*cty*> | *id*

constraint ::= t | **each** (*bty id*; t) { t' }

Users can define first-order pure specification **function**s and resource **predicate**s. The latter syntactically distinguish predicate input and output arguments with a function-like syntax: **predicate** *bty id* ($bty_1\ id_1, \ldots, bty_n\ id_n$) {*specs*} defines resource predicate *id* with input arguments $id_1, \ldots, id_n$ and output type *bty*; the predicate body *specs* is a list of guarded clauses in the form of nested if-then-elses, each clause comprising a list of *conditions* and a return statement that defines the output value.

spec_function_def ::= **function** *bty id*($bty_1\ id_1, .., bty_n\ id_n$) { t }
spec_predicate_def ::= **predicate** *bty id*($bty_1\ id_1, .., bty_n\ id_n$) { *specs* }
specs ::= *spec* | **if** (t) { *spec* } **else** { *specs* } *spec* ::= *conditions* **return** e;

Expressions in resources and constraints range over base types **void** (unit), unbounded **integer**s, fixed-width integers such as `u8` and `i32`, (untyped) **pointer**s, structured types (structs, tuples, and records), collections (lists, sets, and maps), and user-defined **datatype**s.

base_type, *bty* ::= *id* | **void** | **bool** | **integer** | **u** *nat* | **i** *nat* | **pointer** | **struct** *tag* | **tuple**<$bty_1, .., bty_n$>
| { $bty_1\ f_1, .., bty_n\ f_n$ } | **list**<*bty*> | **set**<*bty*> | **map**<bty_1, bty_n> | **datatype** *tag*
datatype_def ::= **datatype** *tag* { $constructor_1, .., constructor_n$ }
constructor ::= C { $bty_1\ f_1, .., bty_n\ f_n$ }

Expressions include CN function application, pattern matching, pointer-shift operations, iterated conjunction over constant ranges (**each**), accesses and pure updates of struct, record, and map values, constructors for tuples and lists, and standard binary and unary operators.

term, t ::= k | *id* | *unop* t | t *binop* t' | *id*($t_1, .., t_n$) | **let** *id* = t; t' | **if** (t_1) { t_2 } **else** { t_3 }
| **match** t { **match_cases** } | **each** (*bty id* : *int_range*; t) | (*bty*) t | &*id* | **array_shift**<*cty*>(t_1, t_2)
| **member_shift**<*tag*>(t, f) | **good**<*cty*>(t) | **sizeof**<*cty*> | **offsetof** (*tag*, f) | t[t'] | t[$t_1 : t'_1, \ldots, t'_n : t_n$]
| { $f_1 : t_1, \ldots, f_n : t_n$ } | t.*id* | { $f_1 : t_1, \ldots, f_n : t_n$..$t$ } | C { $f_1 : t_1, .., f_n : t_n$ } | [$t_1, .., t_n$]
| **default**<*bty*>
match_case ::= *pattern*=> { t } *pattern* ::= _ | *id* | C { f_1 : $pattern_1, .., f_n$: $pattern_n$ }
unop ::= ! | - *binop* ::= * | / | + | - | == | != | < | > | <= | >= | && | || | ==> *int_range* ::= k_1, k_2

Expressions in CN are pure: they can include identifiers of C function arguments, to refer to their initial values, and the addresses of C variables in scope (function arguments and globals), but can depend on the heap only by reference to take-bound outputs of **Owned**, **Block**, or derived predicates. (The CN implementation lets specification expressions "dereference" owned pointers, e.g. using `*p` following **take** `v = Owned<int>(p);` this is merely surface-level syntactic sugar for v, which we elide here.) Unlike CN proof, runtime execution does not require loop invariants; a Fulminate implementation of this is in progress and omitted here. CN also includes annotations for guiding proof and for specifying lemmas, which are not needed for runtime checking.

2.4.1 CN Specifications as Refinement Types. We recall from Pulte et al. [41] how CN specifications relate to mathematically presented refinement types. Absent any use of **take** `id = resource`, CN requires and ensures specifications correspond to types with constraints in the decidable fragment of SMT solvers, following the liquid types discipline of Rondon et al. [44]. Recalling and updating an example from Pulte et al. [41], we start with a pure function that increments a signed integer. To avoid undefined behaviour in C, it must have the pre-condition that its value is strictly less than the maximum value of a signed integer. We see that the corresponding refinement type for this function translates and binds computational C arguments using Πi, and puts constraints on the left of an implication arrow $\Rightarrow$. For ensures clauses, the type binds the computational return using a $\Sigma return$, and puts constraints on the left of a conjunction $\wedge$. Lastly, it terminates types with I.

```
signed int incr(signed int i)
/*@ requires i < MAXi32();
    ensures return == i + 1i32; @*/
{ return i + 1; }
```

$$\Pi i : \texttt{i32}.\ i < \texttt{MAXi32()} \Longrightarrow$$
$$\Sigma return.\ return = i + \texttt{1i32} \wedge I$$

When resources enter the picture, types also include logical variables, separating implications, and separating conjunctions. We omit constraints about pointer-alignment to focus on how the syntax relates to more traditional presentations of separation logic and refinement types. We adjust the previous example to now increment an integer accessed via a *pointer* rather than the value itself. To do so, we take ownership of the pointer in the requires clause so we may dereference and write to it, and return that ownership in the ensures clause because we do not free it. The type for the requires clauses puts resources on the left of separating implications $-\!\!*$, and introduces and binds their outputs immediately before the resources with a $\forall$. The refinement type then proceeds on to the ensures clauses; it puts resources on the left of separating conjunctions $*$, and introduces and binds their outputs immediately before with an $\exists$. Because CN syntax guarantees correct modes via variable scoping, $\forall$ and $\exists$ can be *inferred* based solely on terms derived from inputs.

```
signed int incr_ptr(signed int *p)
/*@ requires take i = Owned(p);
             i < MAXi32();
    ensures  take i2 = Owned(p);
             return == i2;
             i2 == i + 1i32; @*/
{ return ++*p ; }
```

$$\Pi p : \texttt{pointer}.$$
$$\forall i : \texttt{i32}. p \mapsto i \ -\!\!*$$
$$i < \texttt{MAXi32()} \Longrightarrow$$
$$\quad \Sigma return.\ \exists i2 : \texttt{i32}.\ p \mapsto i2\ *$$
$$\quad return = i2 \wedge$$
$$\quad i2 = i + \texttt{1i32} \wedge$$
$$\quad\quad I$$

2.4.2 Executability. The specification language design includes several restrictions of classic separation logic, originally to enable reliable inference, that turn out to be beneficial for runtime checking. (1) Instead of general quantifiers, **take** bindings restrict quantification to just the *outputs* of a resource. These can be automatically inferred in proof, and computed in runtime execution: inputs are arguments needed to make predicates precise (e.g. for **Owned**, the pointer) whereas outputs can be derived from inputs and the owned memory (e.g. for **Owned**, the pointee). For instance, given a concrete `p`, the output `x` in `take x = Owned<int>(p)` can be computed by reading the heap at `p`. In predicate definitions users specify how outputs can be computed from inputs and the owned memory (using **return**). (2) Instead of general disjunction, predicate definitions use if-then-else with guarded clauses (similar to RefinedC [45]). This allows the type inference, in proof, and the runtime checking to determine without backtracking which clause of a resource predicate applies (if any). (3) Instead of general, separating or non-separating, conjunction, function specifications and predicate definition clauses are phrased as "flat" lists of *conditions*.

In addition to these restrictions, previously already imposed for proof, we further restrict quantifiers: the guard expression of an **each** iterated resource or universally quantified constraint has to include lower and upper bounds; i.e. it has to have the shape **each**(*bty id*; e_1<=*id* && *id*<e_2 && ...){ ... } (or similar) so that Fulminate can translate it to the C for-loop the CN syntax alludes to. Note that e_1 and e_2 do not have to be constant, but can be arbitrary expressions, including heap-dependent expressions referring to **take**-bound resource outputs, where the syntactic input/output discipline described above ensures their computability at runtime.

In summary, the restrictions of classic separation logic Fulminate relies on are:

(1) Existential quantifiers are restricted to resource outputs, computable from their inputs and owned memory; this ensures precise predicates.
(2) Predicate definitions use if-then-else instead of regular disjunction.
(3) Specifications do not support general conjunction (separating or non-separating).
(4) Universal quantifiers have upper and lower bounds.

3 Formalisation

To clarify the Fulminate reified ghost state design, and how that relates to explicit ownership passing, in this section we formalise a fragment of Fulminate for a small "miniCN" language. Previous work by Makwana [41, §4] formalised and proved type safety for "kernel CN", a calculus in A-normal form. That was based on the internal "Core" language of Cerberus [32–34]: Cerberus elaborates C source to Core, and the CN proof tool uses the result of that elaboration. Fulminate, on the other hand, is a C+CN source to C source translation, so it is more instructive to use a calculus close to C+CN source. In this section we just highlight the most interesting aspects; the full formalisation and proofs are available online [5].

3.1 MiniCN Base and Specification Language Syntax

The base language is roughly a fragment of C, chosen to illustrate some key aspects of the instrumentation while remaining manageable; its grammar is below. We include global variables, C's mutable function parameters, and block-scoped local variables, and early return, but omit loops and other control flow crossing block boundaries. The base memory object model is just a finite partial function from natural-number addresses to natural-number values. We effectively restrict to the C types $ctype$, $\tau ::= \mathbf{int} \mid \tau*$, though these types are not used in the semantics. We include address-of $\&id$ and dereferencing $*e$, so one can for example pass the address of a function parameter or local to a callee, but simplify C's lvalue language. We omit all structured data (C arrays, structs, unions, pointer arithmetic, and representation-byte accesses), for simplicity. Structured data adds some complexity to the actual implementation, but does not affect ownership checking in fundamental ways. We combine statements and expressions into a single grammar. Base language programs just consist of a collection of global-variable and function definitions, one of which should be `main`.

$declaration,\ decl ::= \tau\ id;$ $\quad lvalue\_expression,\ le ::= n \mid id \mid *le$ $\quad block ::= \{\ decls\ e\}$
$expression,\ e ::= n \mid id \mid \&id \mid uope \mid *e \mid !e \mid e_1\ bop\ e_2 \mid le = e \mid \mathbf{if}\ (e)\ e_1\ \mathbf{else}\ e_2 \mid e; e' \mid f(e_1, .., e_k) \mid block$
$\mid \mathbf{return}\ e$
$function\_definition,\ func ::= \tau\ f(\tau_1\ id_1, .., \tau_k\ id_k) function\_spec\ block;$
$program,\ P ::= spec\_decls\ decls\ func_1\ ..\ func_j$

We then add a fragment of the CN specification language described in §2.4 to this base: function pre- and post-conditions, which are CN predicates $spec$ that can involve recursive predicate definitions and pure-function definitions; the latter with bodies from a pure-term grammar t.

$function\_spec ::=$ `/*@` $\mathbf{requires}\ spec_1\ \mathbf{ensures}\ ret.spec_2$ `@*/`
$spec ::= \mathbf{return}\ t \mid p(t_1, .., t_i) \mid \mathbf{assert}\ (t); spec \mid \mathbf{let}\ id = t; spec \mid \mathbf{take}\ id = spec; spec'$
$\mid \mathbf{if}\ (t)\{spec_1\}\ \mathbf{else}\ \{spec_2\}$
$p ::= \mathbf{Owned} \mid id$
$spec\_function\_def ::= \mathbf{function}\ bty\ f(bty_1\ id_1, .., bty_k\ id_k)\{t\}$
$spec\_predicate\_def ::= \mathbf{predicate}\ bty\ p(bty_1\ id_1, .., bty_k\ id_k)\{spec\}$
$spec\_term,\ t ::= v \mid \{t\} \mid id \mid suop\ t \mid t_1\ sbop\ t_2 \mid \mathbf{if}\ (t)\ t_1\ \mathbf{else}\ t_2 \mid f(t_1, .., t_i)$

3.2 MiniCN Operational Semantics

To have a clear correspondence between the MiniCN formalisation and the actual Fulminate implementation, so that one can see what Fulminate is testing and when, we give the former a dynamic semantics as a small-step abstract machine, with an explicit stack S of stack frames F for function and block invocations, environments giving the memory locations of mutable variables (rather than substituting them out), and a single heap H, which is just a function from $\mathbb{N}$ to $\mathbb{N}$. This makes the top-level small steps of the base-language semantics reasonably close to the behaviour of a standard C implementation (at least when compiling without optimisation), and one can identify the start and end of each allocation's lifetime.

3.2.1 Base Language Semantics. For the base language, the main judgement is, in the context of a program P and global-variable environment E_g, that configuration c takes a reduction step to c'.

$$P, E_g \vdash c \longrightarrow c'$$

A configuration consists of an expression, stack, and heap, along with resource-passing and implementation ghost-state instrumentation that we return to below:

configuration, $c ::= \langle e, S, H, R, G \rangle$

A stack is just a list of stack frames, each either of function or block kind; stack frames contain the expression continuation C (just a list of expression atomic evaluation contexts) and the environment E for their local variables, mapping identifiers to the memory locations where they are allocated. Function stack frames contain the name f of the function, a specification language term value environment V, and resource-passing instrumentation.

stack_frame_kind, $K ::= \texttt{Func}(f, V, R) \mid \texttt{Block}$
stack_frame, $F ::= \langle K, C, E \rangle$

3.2.2 Extensions with CN Predicate Checking. We extend the base language semantics with dynamic checking of MiniCN assertions, and with dynamic semantics for the MiniCN specification language, in two ways:

(1) expressed in terms of explicit resource passing, as introduced informally at the start of Section 2.3, and
(2) expressed in terms of ghost state mapping each memory address to the stack depth (if any) that currently owns it, modelling the Fulminate implementation, as described in the later part of Section 2.3.

We describe the two side-by-side (adding both to the base language semantics), to make it easy to see the correspondence, and in the next subsection prove that they are equivalent.

The resource-passing semantics uses sets of resources R, which are essentially just finite subsets of the memory addresses $\mathbb{N}$. The ambient resources for expression execution are in the R component of configurations, and each C function stack frame includes the resources of the *caller* not required by the currently executing function. That is to say, the stack frame includes resources which are *framed* in the separation-logic sense, so that they may be restored (alongside any post-condition resources) into the configuration upon function return.

The implementation semantics uses ghost ownership mappings G, which are finite partial functions from memory addresses $\mathbb{N}$ to stack depths $\mathbb{N}$. The current stack depth is the number of function stack frames in the stack, with `0` as the notional stack depth for global variables. There is just a single such mapping, in the G component of configurations.

The initial configuration for a program allocates the globals, initialises them to `0`, takes an empty stack, and invokes the `main` function. In the resource-passing semantics, we initialise R to

the footprint of the globals, while in the implementation semantics, we initialise G to map that footprint to `0`.

$$\frac{\begin{array}{l}P = \mathit{spec\_decls}\ \tau_1\ \mathit{id}_1\,;\ ..\ \tau_i\ \mathit{id}_i\,;\ \mathit{func}_1\ ..\ \mathit{func}_j \\ E_{\mathrm{g}} = \mathit{id}_1 \mapsto n_1,\ ..,\ \mathit{id}_i \mapsto n_i \\ S = \mathrm{emp} \\ H = n_1 \mapsto \texttt{0} * ..\ * n_i \mapsto \texttt{0} \\ R = \texttt{Owned}\,(n_1) * ..\ * \texttt{Owned}\,(n_i) \\ G = n_1 \mapsto \texttt{0} * ..\ * n_i \mapsto \texttt{0}\end{array}}{P \longrightarrow (P, E_{\mathrm{g}} \vdash \langle \texttt{main}(),\ S,\ H,\ R,\ G\rangle)}\ \textsc{prog\_init}$$

An R-value read of a C variable *id* first finds its location, in an environment of an enclosing block or function stack frame up to the first function stack frame (inclusive), or otherwise in the global-variable environment (the R-value coercion), and rewrites it to an explicit dereference of that address:

$$\frac{\mathrm{lookup}\ E_{\mathrm{g}}\ S\ \mathit{id} = n}{P, E_{\mathrm{g}} \vdash \langle \mathit{id},\ S,\ H,\ R,\ G\rangle \longrightarrow \langle *n,\ S,\ H,\ R,\ G\rangle}\ \textsc{var}$$

(L-value identifier lookup and taking the address of a variable &*id* are similar.)

A dereference of a concrete address involves a runtime check: in the resource-passing semantics, it has to check that the address is in the ambient resources R, while in the implementation semantics, it has to check that the address is mapped by G to the current stack depth – as C functions do not automatically have access to the local variables of their caller. Then it just finds the heap value of that address:

$$\frac{\begin{array}{l}\mathrm{runtime\_check\_res}\,(n \in R) \\ \mathrm{runtime\_check\_imp}\,(G(n) = \mathrm{stackdepth}(S)) \\ H(n) = n'\end{array}}{P, E_{\mathrm{g}} \vdash \langle *n,\ S,\ H,\ R,\ G\rangle \longrightarrow \langle n',\ S,\ H,\ R,\ G\rangle}\ \textsc{unop\_dereference}$$

(L-value indirection, and assignment, are similar)

Function entry and block entry, and function return and block exit, are interestingly similar and different. Functions and blocks both introduce new mutable local variables for their syntactic scope: the function parameters and block-scoped local variables respectively (recall that in C function parameters are mutable, and that the lifetime of a local variable is that of its enclosing block). However, on function entry, the environment of the caller and its parents becomes irrelevant (they are no longer in scope, and lookup does not recurse into them), and the callee should only gain ownership that is explicitly passed, by any **take**s in the function precondition. On block entry, on the other hand, the new local variables are effectively added to the current environment, and those resources are implicitly added to the ambient resources.

A function call of f finds the function definition in the program P, creates a new environment E' for the function parameters, creates new allocations H' for the function parameters, creates a new specification pure-values environment V, checks the precondition *spec* holds, and executes the function body (*block*) in the new configuration. In the process of checking the precondition, the rule also (a) checks resources required by the precondition are in the ambient resources R and puts them in R'_1, leaving those to be framed on the stack in R'_2; (b) checks resources required by the precondition are mapped to stackdepth(S) in G and *increments* their mappings in G'; and (c) updates the specification pure-values environment to V', so that the initial values of arguments upon function entry and any pure-values bound using **let** *id* and **take** *id* are available when

checking the postcondition $ret.\ spec'$ upon function return.

$$\frac{\begin{array}{l}P(f) = \tau\ f(\tau_1\ id_1,\ ..,\ \tau_k\ id_k)\texttt{/*@ requires } spec \texttt{ ensures } ret.spec'\ \texttt{@*/ } block\texttt{;}\\ E' = id_1 \mapsto m_1,\ ..,\ id_k \mapsto m_k\\ H' = m_1 \mapsto n_1 * .. * m_k \mapsto n_k\\ V = id_1 \mapsto n_1,\ ..,\ id_k \mapsto n_k\\ P, E_g, H, \texttt{inc}, \text{stackdepth}(S) \vdash \langle spec,\ V,\ \text{emp},\ R,\ G\rangle \longrightarrow^* \langle \cdot,\ V',\ R'_1,\ R'_2,\ G'\rangle\\ R'' = \texttt{Owned}\ (m_1) * .. * \texttt{Owned}\ (m_k)\\ S' = \langle \texttt{Func}(f, V', R_2), \text{emp}, E'\rangle :: S\\ G'' = m_1 \mapsto \text{stackdepth}(S') * .. * m_k \mapsto \text{stackdepth}(S')\end{array}}{P, E_g \vdash \langle f(n_1,\ ..,\ n_k),\ S,\ H,\ R,\ G\rangle \longrightarrow \langle block,\ S',\ H * H',\ R'_1 * R'',\ G' * G''\rangle}\ \textsc{call}$$

A block entry likewise has to create new environment E' and allocations H', for the block-scoped local variables, but these are simply added to those already available (respectively by creating a new **Block** frame, which lookup will recurse through, and by starring onto the existing H). The resource-passing and implementation instrumentation follow suit, with the latter using the current stack depth (which is unaffected by the **Block** frame).

$$\frac{\begin{array}{l}block = \{\ \tau_1\ id_1\texttt{;}\ ..\ \tau_i\ id_i\texttt{;}\ e\}\\ E' = id_1 \mapsto n_1,\ ..,\ id_i \mapsto n_i\\ H' = n_1 \mapsto \texttt{0} * .. * n_i \mapsto \texttt{0}\\ R' = \texttt{Owned}\ (n_1) * .. * \texttt{Owned}\ (n_i)\\ G' = n_1 \mapsto \text{stackdepth}(S) * .. * n_i \mapsto \text{stackdepth}(S)\end{array}}{P, E_g \vdash \langle block,\ S,\ H,\ R,\ G\rangle \longrightarrow \langle e,\ \langle \texttt{Block}, \text{emp}, E'\rangle :: S,\ H * H',\ R * R',\ G * G'\rangle}\ \textsc{block\_start}$$

For a function return, the base-language semantics discards the **Func** stack frame, along with all its **Block** frames, and removes the footprint of the function parameters and block-scoped locals from the heap, with the resource-passing and implementation instrumentation doing the same. It updates the specification pure-value environment with the returned value ($V, ret \mapsto n$) (so the postcondition can talk about it), and checks the post-condition $spec'$ holds. This: (a) in the resource-passing semantics, checks resources ensured by the postcondition are exactly the ambient ones R_1 (no leaks); (b) in the implementation instrumentation, (b.1) checks resources ensured by the postcondition are mapped to stackdepth(S) in G' and *decrements* their mappings in G''; and (b.2) checks for leaks by iterating through all of G'' and ensuring all ghost mappings are bounded by stackdepth(S'), one less than stackdepth(S); and (c) updates the specification pure-values environment with any pure-values bound using **let** id and **take** id.

$$\frac{\begin{array}{l}S = \langle \texttt{Block}, C_1, E_1\rangle :: .. :: \langle \texttt{Block}, C_i, E_i\rangle :: \langle \texttt{Func}(f, V, R_2), C, E\rangle :: S'\\ H' = H \setminus \text{dom}\ (E_1,\ ..,\ E_i, E)\\ R_1 = R \setminus \text{dom}\ (E_1,\ ..,\ E_i, E)\\ G' = G \setminus \text{dom}\ (E_1,\ ..,\ E_i, E)\\ P(f) = \tau\ f(\tau_1\ id_1,\ ..,\ \tau_k\ id_k)\texttt{/*@ requires } spec \texttt{ ensures } ret.spec'\ \texttt{@*/ } block\texttt{;}\\ P, E_g, H, \texttt{dec}, \text{stackdepth}(S) \vdash \langle spec',\ (V, ret \mapsto n),\ \text{emp},\ R_1,\ G'\rangle \longrightarrow^* \langle \cdot,\ V',\ R_1,\ \text{emp},\ G''\rangle\\ \text{runtime\_check\_imp}\ (\text{stackdepth}(S') \vdash G'')\end{array}}{P, E_g \vdash \langle \texttt{return}\ n,\ S,\ H,\ R,\ G\rangle \longrightarrow \langle n,\ S',\ H',\ R_1 * R_2,\ G''\rangle}\ \textsc{return}$$

A block end just has to discard the **Block** stack frame and the footprint of its locals.

$$\frac{}{P, E_g \vdash \langle n,\ (\langle \texttt{Block}, \text{emp}, E\rangle :: S),\ H,\ R,\ G\rangle \longrightarrow \langle n,\ S,\ H \setminus \text{dom}\ (E),\ R \setminus \text{dom}\ (E),\ G \setminus \text{dom}\ (E)\rangle}\ \textsc{block\_end}$$

3.3 Dynamic Semantics of the MiniCN Specification Language

The above function call and return rules appeal to semantics for CN specifications, to check the ownership and pure-value aspects of the **requires** precondition and **ensures** postcondition, and to transfer ownership appropriately. The syntactic form of CN specifications means they can be

executed as recursive functions. In this formalisation, we capture this with a small-step dynamic semantics, while in the Fulminate implementation, they are translated to plain C, which is compiled and executed with the conventional C toolchain.

The specification dynamic semantics is expressed as a judgement:

$$P, E_g, H, \mathit{incdec}, \mathit{sd} \vdash \langle \mathit{spec},\ V,\ R_1,\ R_2,\ G\rangle \longrightarrow \langle \mathit{spec}',\ V',\ R_1',\ R_2',\ G'\rangle$$

Specifications may contain references to user-defined predicates, the definitions of which are looked up in the program P. Their execution only reads from, but does not write to the heap H, which remains constant. Specifications may also refer to and take ownership of globals, and so E_g provides their address. Specific to ghost ownership checking, we also have *incdec* and *sd*. The construct *incdec* parameterises over the slightly different modes needed for executing specifications in **`requires`** and **`ensures`** contexts. That is, **`requires`** transfers ambient resources *up* the stack and so **`inc`**rements their ghost ownership mappings; **`ensures`** transfers ambient resources *down* the stack and so **`dec`**rements their ghost ownership mappings. The current stack-depth *sd* is used to check that resources *used* during the course of executing the specification are in fact the *ambient* ones at that stack-depth. Similar to globals, specifications are defined with function parameters in scope, but unlike globals, it is their r-values, not their addresses. (Specifications would be confusing to read if function parameters were mutable, and tedious to write if the user had to take ownership of a function parameter to read it.) The pure-value environment V is initialised to this mapping of function parameter r-values, and accumulates further bindings for every **`let`** `id` or **`take`** `id` it executes. In the resource passing semantics, R_1 represents exactly those *used* ambient resources which are going to move up or down the stack. In contrast, R_2 represents the ambient resources from which are *so far* not used, that is not to be transferred to the ambient resources of the callee (and instead, framed on the stack), or *so far* leaking from the caller.

An assertion evaluates by checking that the value v it contains is equal to **`true`**.

$$\frac{\mathrm{runtime\_check\_res}\ (v = \texttt{true}) \qquad \mathrm{runtime\_check\_imp}\ (v = \texttt{true})}{P, E_g, H, \mathit{incdec}, \mathit{sd} \vdash \langle \texttt{assert}\ (v)\,;\mathit{spec},\ V,\ R_1,\ R_2,\ G\rangle \longrightarrow \langle \mathit{spec},\ V,\ R_1,\ R_2,\ G\rangle}\ \textsc{spec\_assert}$$

A **`let`** evaluates by updating the pure-value environment with its value $(V, \mathit{id} \mapsto v)$.

$$\frac{}{P, E_g, H, \mathit{incdec}, \mathit{sd} \vdash \langle \texttt{let}\ \mathit{id} = v\,;\mathit{spec},\ V,\ R_1,\ R_2,\ G\rangle \longrightarrow \langle \mathit{spec},\ (V, \mathit{id} \mapsto v),\ R_1,\ R_2,\ G\rangle}\ \textsc{spec\_let}$$

A call to a user-defined predicate $p(v_1, .., v_k)$, finds the definition in the program, $P(p) = \texttt{predicate}\ \mathit{bty}\ p(\mathit{bty}_1\ \mathit{id}_1, .., \mathit{bty}_k\ \mathit{id}_k)\{\mathit{spec}\}$, constructs a mapping of its formal parameters to the pure specification values with which it was called, pushes this mapping to the top of the specification pure-value environment *stack* $(\mathit{id}_1 \mapsto v_1, .., \mathit{id}_k \mapsto v_k) :: V$ (to ensure lexical scoping), and finally steps to the body of p, *spec*.

$$\frac{P(p) = \texttt{predicate}\ \mathit{bty}\ p(\mathit{bty}_1\ \mathit{id}_1, .., \mathit{bty}_k\ \mathit{id}_k)\{\mathit{spec}\} \qquad V' = (\mathit{id}_1 \mapsto v_1, .., \mathit{id}_k \mapsto v_k) :: V}{P, E_g, H, \mathit{incdec}, \mathit{sd} \vdash \langle p(v_1, .., v_k),\ V,\ R_1,\ R_2,\ G\rangle \longrightarrow \langle \mathit{spec},\ V',\ R_1,\ R_2,\ G\rangle}\ \textsc{spec\_pred\_user}$$

We syntactically require that all calls to predicates are bound to a **`take`**. Therefore, all predicates evaluate underneath the binding of a **`take`**.

$$\frac{P, E_g, H, \mathit{incdec}, \mathit{sd} \vdash \langle \mathit{spec},\ V,\ R_1,\ R_2,\ G\rangle \longrightarrow \langle \mathit{spec}',\ V',\ R_1',\ R_2',\ G'\rangle}{P, E_g, H, \mathit{incdec}, \mathit{sd} \vdash \langle \texttt{take}\ \mathit{id} = \mathit{spec}\,;\mathit{spec}_1,\ V,\ R_1,\ R_2,\ G\rangle \longrightarrow \langle \texttt{take}\ \mathit{id} = \mathit{spec}'\,;\mathit{spec}_1,\ V',\ R_1',\ R_2',\ G'\rangle}\ \textsc{spec\_take\_step}$$

We syntactically require all user-defined predicates must end in a **return** v. This means that eventually, all predicates evaluate to a **return** v, which can then be bound to the binder id of a **take**. Since calls to predicates push on to the pure-value environment stack, returns pop from the top (again, to ensure lexical scoping).

$$\frac{}{P, E_g, H, incdec, sd \vdash \langle \textbf{take}\ id = \textbf{return}\ v; spec,\ V_1 :: V,\ R_1,\ R_2,\ G\rangle \longrightarrow \langle spec,\ (V, id \mapsto v),\ R_1,\ R_2,\ G\rangle}\ \textsc{spec\_take\_return}$$

Recall that **take** `v = Owned(p)` is CN syntax for $\exists v.p \mapsto v$ in separation logic. A call to the primitive **Owned** predicate: (a.1) checks that n is in the ambient resources $\text{runtime\_check\_res}\,(n \in R_2)$; (a.2) removes that resource from the ambient resources $R_2 = R_2' * \textbf{Owned}\,(n)$; (a.3) marks that resource as used $R_1' = R_1 * \textbf{Owned}\,(n)$; (b.1) checks that the ghost ownership mapping is at the correct stack-depth $\text{runtime\_check\_imp}\,(G(n) = sd)$; (b.2) updates its mapping $G' = G_1 * n \mapsto incdec(sd)$ by incrementing or decrementing as per the left of the turnstile; (c) looks up the value pointed to by n in the heap $H(n) = n'$; and (d) steps to that value as **return** n'. For consistency with the evaluation of user-defined predicates, this also pushes an empty environment $\cdot$ to the top of the pure-value environment stack.

$$\frac{\begin{array}{l}H(n) = n' \\ \text{runtime\_check\_res}\,(n \in R_2) \quad R_2 = R_2' * \textbf{Owned}\,(n) \quad R_1' = R_1 * \textbf{Owned}\,(n) \\ \text{runtime\_check\_imp}\,(G(n) = sd) \quad G = G_1 * n \mapsto sd \quad G' = G_1 * n \mapsto incdec(sd)\end{array}}{P, E_g, H, incdec, sd \vdash \langle \textbf{Owned}(n),\ V,\ R_1,\ R_2,\ G\rangle \longrightarrow \langle \textbf{return}\ n',\ \cdot :: V,\ R_1',\ R_2',\ G'\rangle}\ \textsc{spec\_pred\_owned}$$

We demonstrate this semantics using the postcondition shown earlier, assuming a heap such that $H(n) = 0$ and $V = (id \mapsto n, ret \mapsto 0)$, $t = (id' == 0\ \&\&\ ret == id')$.

$$\begin{array}{l}\langle \textbf{take}\ id' = \textbf{Owned}(id); \textbf{assert}\,(t); \cdot,\ V,\ \text{emp},\ \textbf{Owned}\,(n),\ G * n \mapsto sd\rangle \\ \longrightarrow \langle \textbf{take}\ id = \textbf{Owned}(n); \textbf{assert}\,(t); \cdot,\ V,\ \text{emp},\ \textbf{Owned}\,(n),\ G * n \mapsto sd\rangle \\ \longrightarrow \langle \textbf{take}\ id' = \textbf{return}\ 0; \textbf{assert}\,(t); \cdot,\ (\cdot :: V),\ \textbf{Owned}\,(n),\ \text{emp},\ G * n \mapsto \textbf{inc}(sd)\rangle \\ \longrightarrow \langle \textbf{assert}\,(t); \cdot,\ (id' \mapsto 0, V),\ \textbf{Owned}\,(n),\ \text{emp},\ G * n \mapsto \textbf{inc}(sd)\rangle \\ \ldots \\ \longrightarrow \langle \textbf{assert}\,(\textbf{true}); \cdot,\ (id' \mapsto 0, V),\ \textbf{Owned}\,(n),\ \text{emp},\ G * n \mapsto \textbf{inc}(sd)\rangle \\ \longrightarrow \langle \cdot,\ (id' \mapsto 0, V),\ \textbf{Owned}\,(n),\ \text{emp},\ G * n \mapsto \textbf{inc}(sd)\rangle.\end{array}$$

3.4 Correspondence between Resource Passing and Implementation Instrumentation

To prove that that the MiniCN implementation instrumentation correctly checks for ownership, we define a formal relation between it and the resource passing dynamic semantics. We then prove that this is preserved by each step of the dynamic semantics with a proof that the two versions bisimulate each other.

Let S_r be a stack in the resource passing version, and S be a stack in the implementation version. Let R be the ambient resource of a resource passing configuration and G be the implementation ghost state. Let sd be the stack-depth of S_r, and let R_i be the resources in the function frames in the stack S_r, for $0 \le i \le sd - 1$.

Definition 3.1 (Resource passing and implementation instrumentation are related.). $(S_r, R) \sim (S, G)$ iff S is S_r with resources erased from function frames, and $G = \left(\circledast_{i=0}^{sd-1} \circledast_{\text{Owned}(n) \in R_i} n \mapsto i\right) * \left(\circledast_{\text{Owned}(n) \in R} n \mapsto sd\right)$.

This definition formally captures the informal description earlier that the ghost state maps "each memory address to the stack depth (if any) that currently owns it".

Lemma 3.2 ($\sim$ is an invertible relation.). *There exists $\sim'$ such that for all S_r, R, S, G, $(S_r, R) \sim (S, G) \Leftrightarrow (S, G) \sim' (S_r, R)$.*

Proof: §1.1 of the supplementary materials.

This definition can be straightforwardly lifted to entire configurations as follows.

Definition 3.3 (Resource passing and implementation configurations are related.). For resource passing configuration $c_r = \langle e, S_r, H, R \rangle$, and implementation configuration $c = \langle e, S, H, G \rangle$, $c_r \sim c_g$ iff $(S_r, R) \sim (S, G)$.

With these definitions established, we prove that the resource passing and implementation instrumentation versions of the semantics are related initially.

Theorem 3.4 (Initial resource passing and implementation instrumentation are related.). *For all programs P and globals E_g,*

- *if $P \longrightarrow (P, E_g \vdash c_g)$ then there exists a c_r such that $P \longrightarrow (P, E_g \vdash c_r)$ and $c_r \sim c_g$,*
- *if $P \longrightarrow (P, E_g \vdash c_r)$ then there exists a c_g such that $P \longrightarrow (P, E_g \vdash c_g)$ and $c_r \sim c_g$.*

Proof: §1.4 of the supplementary materials.

And we prove that the resource passing and implementation instrumentation versions of the semantics, once related, remain related.

Theorem 3.5 (Resource passing and implementation instrumentation bisimulate each other.). *For all resource passing configurations $c_r = \langle e, S_r, H, R \rangle$, and implementation configurations $c_g = \langle e, S, H, G \rangle$, such that $c_r \sim c_g$,*

- *if $P, E_g \vdash c_r \longrightarrow c_r'$ then there exists a c_g' such that $P, E_g \vdash c_g \longrightarrow c_g'$ and $c_r' \sim c_g'$,*
- *if $P, E_g \vdash c_g \longrightarrow c_g'$ then there exists a c_r' such that $P, E_g \vdash c_r \longrightarrow c_r'$ and $c_r' \sim c_g'$.*

Proof: §1.5 of the supplementary materials.

Corollary 3.6 (Resource passing and implementation instrumentation fail under the same conditions.). *This consists of two aspects.*

- *For all related configurations $\langle e, S_r, H, R \rangle \sim \langle e, S, H, G \rangle$,*
  $\mathsf{runtime\_check\_res}\,(n \in R) \Leftrightarrow \mathsf{runtime\_check\_imp}\,(G(n) = sd)$,
- *For all stacks S, S_r, pure-value environments V, resources R', and instrumentation G,*
  $(S_r, \langle \cdot, V, R', \mathsf{emp} \rangle) \sim_{\mathbf{dec}} (S, u, \langle \cdot, V, G \rangle) \Leftrightarrow \mathsf{runtime\_check\_imp}\,(\mathbf{dec}(\mathsf{stackdepth}(S)) \vdash G)$.

Proof: corollaries in §1.1-3 of the supplementary materials.

The key steps of this proof hinge on the evaluation of specifications in function calls and returns. Whilst the judgement for the specification evaluation abstracts over the choice of **inc** or **dec**, for the proof, it is convenient to handle these two cases separately.

*Definition 3.7 (Resource passing and implementation ghost state are related during **inc** specification evaluation.).* $(S_r, R', R'') \sim_{\mathbf{inc}} (S, G)$ iff S is S_r without resources in function frames and $G = \left(\ast_{i=0}^{sd-1} \ast_{\mathsf{Owned}(n) \in R_i} n \mapsto i\right) * \left(\ast_{\mathsf{Owned}(n) \in R''} n \mapsto sd\right) * \left(\ast_{\mathsf{Owned}(n) \in R'} n \mapsto sd + 1\right)$.

This relation is used in the proof that call preserves $\sim$. Specifically, we show that $c_r \sim c_g$ implies the initial conditions for the **inc** specification evaluation – $(S_r, \mathsf{emp}, R) \sim_{\mathbf{inc}} (S, G)$ – and that the final conditions $(S_r, R_1', R_2') \sim_{\mathbf{inc}} (S, G')$ imply $(S_r', R_1' * R'') \sim (S', G' * G'')$ where $S_r' = \langle \mathsf{Func}(f, V', R_2''), \mathsf{emp}, E' \rangle :: S_r$.

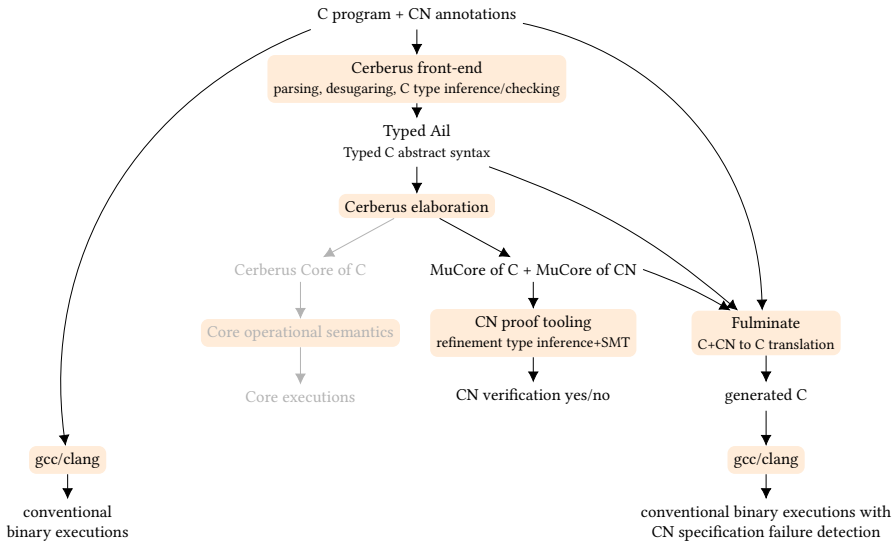

Fig. 1. Fulminate architecture

Definition 3.8 (Resource passing and implementation ghost state are related during ***dec*** *specification evaluation.).* $(S_r, R', R'') \sim_{\mathbf{dec}} (S, u, G)$ iff S is S_r without resources in function frames and $G = \left(\ast_{i=0}^{sd-1} \ast_{\mathsf{Owned}(n) \in R_i} n \mapsto i\right) * \left(\ast_{\mathsf{Owned}(n) \in R'} n \mapsto sd - 1\right) * \left(\ast_{\mathsf{Owned}(n) \in R''} n \mapsto sd\right)$ and $u = \{n \mid \mathsf{Owned}\,(n) \in R'\}$.

This relation is used in the proof that RETURN preserves $\sim$. Specifically, we show that $c_r \sim c_g$ implies the initial conditions for the **dec** specification evaluation – $(S_r, \mathrm{emp}, R) \sim_{\mathbf{dec}} (S, G)$ – and that the final conditions $(S_r, R, \mathrm{emp}) \sim_{\mathbf{dec}} (S, G')$ imply $(S'_r, R, R_{sd-1}) \sim (S', G' * G'')$ where $S_r = \langle \mathsf{Block}, C_1, E_1 \rangle :: \ldots :: \langle \mathsf{Block}, C_i, E_i \rangle :: \langle \mathsf{Func}(f, V', R_{sd-1}), \mathrm{emp}, E' \rangle :: S'_r$.

4 Implementation

4.1 Architecture

CN builds on the Cerberus C semantics by Memarian et al. [32–34], benefitting from its coverage of intricate C features and well-validated semantics to handle much of real C, not some idealised C-like language. It uses the Cerberus front-end to parse, desugar, and C-type-check, producing the *Typed Ail* intermediate language, and a variant of the Cerberus elaboration to translate that *MuCore*, capturing many aspects of the C dynamic semantics. The CN proof tooling – its separation-logic refinement type inference – works over MuCore, using an SMT solver.

Fulminate reuses some of this infrastructure but not all, as it is a C+CN to C source to source translator. As shown in Fig. 1, it uses the same front-end to Typed Ail, and the elaboration of CN specifications to their MuCore form, to generate executable versions of CN specifications as Typed Ail; it then injects a pretty-print of this generated code into the appropriate places in a copy of the original source file (and in other new files). Injecting into the original source, rather than simply pretty-printing modified Typed Ail, keeps the structure, layout, and comments of the original, making the generated code easier to navigate when necessary. The result can be built and executed with standard C compilers such as gcc and clang.

The translation has to handle aspects arising from the CN specification language, which as we saw in §2.4 includes a substantial first-order functional language, from C, with its many intricacies, and from the various ways in which CN specifications can be intertwined with the surrounding C and CN code: they can mention function parameter and return values, refer to C types, and, in arguments to **`Owned`**, refer to the C heap. They can also refer to CN datatype, predicate, and function definitions.

4.2 Basic Translation Scheme

The basic translation scheme is simple, and is what one would expect. CN specifications appear in C chiefly in pre- and post-conditions. For a minimal example with just a pure CN precondition, that requires the function argument to be zero:

```
void f(int x)  /*@ requires x==0i32; @*/
{
  return;
}
```

Fulminate injects an executable C version of the check `x==0`: reading the function argument, converting it to the Fulminate C representation of the CN mathematical value thereof (which here is essentially a no-op), computing the appropriate equality, and reporting an error if that fails. Simplified slightly for presentation, the generated code is:

```
void f(int x) /*@ requires x==0i32; @*/
{
  cn_bits_i32* x_cn = convert_to_cn_bits_i32(x);
  cn_assert(cn_bits_i32_equality(x_cn, convert_to_cn_bits_i32(0)));
  return;
}
```

The injected code computes the abstract value of each function argument on entry, and after that the specifications just talk about pure values. CN specifications can also refer to any **`return`** value of a function, and in C one can **`return`** from any point (perhaps inside blocks or loops), so the injected code computes the abstract value for any **`return`** and then **`goto`**s to an added epilogue that checks the postcondition.

CN predicate and function definitions, which are embedded in C source files in top-level comments `/*@ ... @*/`, are translated to C function definitions at those points.

4.3 Translation of CN Types and Type Definitions

A conventional compiler would typically either be free to use its own binary representations or conform to a standard ELF representation. Here, to make the generated code readable (for ease of debugging dynamic CN specification failures), and to exploit the type checking of the C target language as much as possible (limited though that is), for ease of debugging the translator, we map CN types onto corresponding C types. CN values are uniformly mapped into pointers to in-memory structs, e.g. the CN `x` above, which has CN type `i32` in the specification (for the pure value of the C function argument), is translated to `x_cn`, of C type **`struct`** `cn_bits_i32*`.

A C struct definition, e.g. **`struct`** `s { signed int x;};`, is translated to the corresponding instrumentation C struct definition **`struct`** `s_cn { cn_bits_i32* x; };`, together with a conversion function from the former to the latter, and an equality function for the latter.

CN lists, sets, and maps are translated into C representations using a small Fulminate library, with a hashtable for maps.

CN datatype definitions are translated into C tagged unions. For example, this `seq` type of mathematical lists of `i32` integers:

```
datatype seq { Seq_Nil {}, Seq_Cons {i32 head, datatype seq tail} }
```

is translated into the C tagged union:

```
enum seq_tag {
  SEQ_NIL,
  SEQ_CONS };
struct seq_nil { };
struct seq_cons {
  struct seq* tail;
  cn_bits_i32* head; };

union seq_union {
  struct seq_nil* seq_nil;
  struct seq_cons* seq_cons; };

struct seq {
  enum seq_tag tag;
  union seq_union u; };
```

4.4 Translation of the CN Expression Language

The translation of CN expressions has to take into account the fact that the C target distinguishes expressions and statements, and that they can be used in various kinds of contexts. We use a destination-passing style, loosely analogous to that of Shaikhha et al. [49], and OCaml GADTs for this. Our implementation handles four kinds of statement language context: `AssignVar x`, `Return`, `Assert` and `PassBack`. These can be used flexibly to propagate expressions throughout the translation. CN expressions include pattern matching, for which we generate C implementations following the matrix algorithm of Maranget [31]. For the CN integer operations, we generate implementations for each type (width and signedness) at which they are used, using macros that take the CN and C types. The CN iterated separating conjunction **each**$(bty\ x;\ t)\{p(t_1,\ldots,t_n)\}$ is translated to a loop over its range, which for testing (though not for CN proof) must be bounded.

4.5 Ownership

The implementation of ownership checking follows the general scheme outlined in §2.3 and formalised in §3: it maintains reified ghost state recording the stack depth at which each byte of memory is owned, as a global variable. Currently this is a simple hashtable from virtual addresses to that metadata, for simplicity; many optimisations are obviously possible.

4.5.1 Updating Ownership Metadata on Object Creation/Deletion. For object creation, for globals, function parameters, and block-scoped local variables, Fulminate injects code to update this mapping at the lifetime starts: in `main`, the function preamble, and at block-scoped declarations. For function parameters and block-scoped locals, it injects code to delete the corresponding ownership, in the (CN-introduced) function epilogue and at block-end.

C has various forms of unstructured control flow: early **`return`**, **`break`**, **`continue`**, and **`goto`**. All of these can exit one or more block scopes, which also requires deleting any associated mappings (for block-scoped mappings, only those that have been reached by the early block exit point). Exotically, C **`goto`** can also be used to jump *into* block scopes, which requires creating mappings for all the locals of the entered blocks. Fulminate uses an analysis of the Typed Ail to compute the ownership that must be created and destroyed for each of these, so it can inject code to do so just before the control-flow change – though currently only **`return`** and **`break`** are fully implemented.

For C locals whose address is not taken, there is an obvious potential optimisation to omit ownership recording and checking, but at present we uniformly track ownership for all allocations.

4.5.2 Updating and Checking Ownership Transfer on Function Call and Return. The ownership checks and transfer on a function call and return, for any **`take`**s in **`requires`** and **`ensures`** clauses,

was outlined in those previous sections: in brief, the injected code executes the (translation of) the CN predicates, as compiled to (possibly recursive) C functions. This ends up with invocations of the primitive **Owned** predicate. For example, the translation of:

```
void f(int *p)
/*@ requires take v = Owned<int>(p);
    ensures take w = Owned<int>(p); @*/
{ }
```

which requires and ensures ownership of the target of p, computes the CN representation of that pointer value (basically a no-op):

```
  cn_pointer* p_cn = convert_to_cn_pointer(p);
```

and then for the precondition and postcondition makes two calls to the CN implementation of **Owned** for that type, parameterised on whether in GET (**inc**) mode:

```
cn_bits_i32* v_cn = owned_signed_int(p_cn, GET);
```

or PUT (**dec**) mode:

```
cn_bits_i32* w_cn = owned_signed_int(p_cn, PUT);
```

Such functions are generated for all the required types; they check and update the reified ghost-state finite map from addresses to ownership depth, as described earlier.

4.5.3 Checking Ownership at Memory Accesses. Fulminate also (optionally) instruments all memory accesses, to check that their footprint is in the ambient ownership. This involves instrumenting all instances of lvalue conversions, assignments and postfix increment and decrement expressions.

We experimented using existing AddressSanitizer compiler instrumentation for accesses, instead of injecting instrumentation into the source. At first sight this is promising: there are flags, albeit some undocumented, which together make it instrument most memory accesses with calls to instrumentation functions (taking the address and size of the access) which we could replace. However, it turned out that C compound literals were not consistently instrumented, and avoiding AddressSanitizer instrumentation of the Fulminate instrumentation was awkward. Moreover, depending only on Cerberus rather than also on a specific compiler simplifies use. For ease of use we also wanted to avoid Clang- or GCC-specific extensions.

4.6 Dealing with the C

The fact that CN is embedded in C, with C variables and type names usable within CN, including the ability to talk about pure values corresponding to C **struct** types, is crucial for usability. The surface language shields the user from the underlying scoping complexity, but Fulminate has to correctly handle it in the generated code: all the CN-generated C types and C functions need to be visible in all the translated code. At present we inject prototypes at the original definition points, and generate definitions into a single cn.c file. Our examples have manual includes, which we handle, but not multiple-file compilation units, which we do not at present.

4.7 Design for Usability

Generated code is notoriously often hard to work with, but the above care to make it readable C, including the preservation of the original source-file layout and comments, and insertion of comments identifying the different kinds of injected code, pays off: one can make sense of the generated code, when debugging the translation and, we believe, when debugging runtime checking failures. One can reasonably use standard debuggers such as gdb or lldb on the generated code.

We have also devoted considerable effort to error-message reporting, both for CN base typechecking and for runtime checking failures. The latter report both the original C+CN source location and the generated-code location.

5 Examples

5.1 Queues

We illustrate Fulminate runtime testing first on a relatively simple integer queue example. Though informally simple to describe as a first-in-first-out data structure, giving a formal specification that correctly handles ownership is nontrivial. We first briefly explain the key aspects of a correct CN specification, and then move on to how realistic mistakes (ones actually made during development) can be diagnosed from runtime assertion failures in the instrumented execution.

A **struct** int_queue is a pair of two pointers, front and back, pointing respectively to the head and to the last element of a linked list. The nodes of the linked list are **struct** int_queueCells which contain the integer payload first and a pointer to the subsequent node next.

```
struct int_queue {
  struct int_queueCell* front;
  struct int_queueCell* back; };
```

```
struct int_queueCell {
  int first;
  struct int_queueCell* next; };
```

Predicate IntQueuePtr takes a pointer to a **struct** int_queue. It requires ownership of this, with **take** Q = **Owned**<..>(..), so that it may be dereferenced and its members accessed. The asserted invariant (**assert** (both || neither)) describes the two valid possibilities: either the queue is empty when *both* its pointers are NULL (**let** both = ..), or it is non-empty when *neither* are NULL (**let** neither = ..). It then defers to **take** L = IntQueueFB(..) to traverse/claim ownership over any linked queue cells, and produce a ghost value representing the queue as a linked list.

```
/*@ predicate (datatype seq)
      IntQueuePtr (pointer q) {
        take Q = Owned<struct int_queue>(q);
        let both = is_null(Q.front)
                   && is_null(Q.back);
        let neither = !is_null(Q.front)
                      && !is_null(Q.back);
        assert (both || neither);
        take L = IntQueueFB(Q.front, Q.back);
        return L;
      } @*/
```

```
/*@ predicate (datatype seq)
      IntQueueFB (pointer front, pointer back) {
        if (is_null(front)) {
          return Seq_Nil{};
        } else {
          take B = Owned<struct int_queueCell>(back);
          assert (is_null(B.next));
          take L = IntQueueAux (front, back);
          return snoc(L, B.first);
        }
      } @*/
```

Predicate IntQueueFB takes as its arguments two potentially null pointers. Since it is used only by IntQueuePtr, it tests only front to check if it is null before returning the empty list. Otherwise, it takes ownership of the *back* of the queue (**take** B = **Owned**<..>(..)), and asserts it has no next node. Taking ownership directly from the top here is critical for pushing on to the end of the queue in constant time. The predicate then recursively claims ownership of the rest of the queue (**take** L = IntQueueAux(..)), using the auxiliary predicate to represents a segment of a linked list from front to back.

Using these, and standard mathematical list functions, we can define the interface and specifications for the queue's three operations: IntQueue_empty creates an empty queue (with the invariant in place); IntQueue_pop removes an integer from the front of a non-empty queue; IntQueue_push

adds an element to the back of a queue. As we will see later, the last one has the most complex ownership reasoning.

```
struct int_queue* IntQueue_empty()
/*@ ensures take ret = IntQueuePtr(return); ret == Seq_Nil{}; @*/

int IntQueue_pop (struct int_queue *q)
/*@ requires take before = IntQueuePtr(q); before != Seq_Nil{};
    ensures  take after  = IntQueuePtr(q); after == tl(before); return == hd(before); @*/

void IntQueue_push(int x, struct int_queue *q)
/*@ requires take before = IntQueuePtr(q);
    ensures  take after  = IntQueuePtr(q); after == snoc (before, x); @*/
```

We omit the definition of some functions which are no-ops in C, but whose pre-conditions assert the value of the queue as lists. When translated from CN+C into C by Fulminate, the code tests that any sequence of operations executes and gives observably correct values, and also that the assertions and ownership are correct at every step. We also omit a `main` function that exercises the queue with a sequence of creates, pushes and pops.

If the specification and implementation are consistent, then the instrumented code will run to completion. Otherwise (so long as the test case exercises the code sufficiently), there will be a dynamic error. For example, a copy-paste mistake resulted in `snoc` always returning the empty list. This gives a `CN assertion failed` error, pointing to the `before != Seq_Nil {}` in the pre-condition of `IntQueue_pop`. Since we know we pushed to the stack before calling pop, we can use a debugger to see that `*(queue->back)` is definitely not empty, and so the specification must be wrong to conclude `before == Seq_Nil{}`. Memory management issues such as leaked resources also trigger dynamic errors.

Errors from mistaken predicate definitions give users the ability to incrementally discover general predicate definitions from concrete feedback, in manual counter-example guided refinement. For example, for these queues, it is easy to try claim ownership for the pointer to the last node twice, because it is aliased both from `queue->back`, and from traversing from the `queue->front`. Logs of the instrumented execution show such duplicated ownership, pointing to the place where a specification change is needed.

5.2 CN, VeriFast and Verifiable C Tutorial Examples

The CN Tutorial is a series of 75 examples designed to teach people how to use CN for verifying C programs, by Pulte, Pierce, and Austell [42]. It consists of annotated C files, covering the major features of CN proof mode such as signed and unsigned integer arithmetic, pointers, arrays, datatypes, and dynamic allocation; it culminates with a few more involved examples such as linked lists, stacks with sizes, and the above queues.

For this paper, we have combined some of these (where multiple related functions were split into separate files for verification exercises) into 54 whole programs, and extended those with `main` functions to exercise the code. These are intended to pass testing, occasionally with different ways to prompt failure mentioned in comments. More involved tests have been added for the list, stack and queue data structures.

We also similarly ported a subset of the VeriFast C tutorial, excluding examples for which CN does not support the required features (such as function pointers, polymorphism, concurrency), and a simple hashtable from Verifiable C.

All examples run successfully in Fulminate; additionally, the CN and VeriFast tutorial examples were verified using CN's proof mode.

5.3 pKVM Buddy Allocator

For a more substantial example we apply Fulminate to an allocator implementation from the production pKVM hypervisor [13] for Android. The allocator, an earlier version of which was used as a case study for CN proof [41], lets us demonstrate Fulminate on production systems software that requires subtle ownership invariants for reasoning about computed memory accesses; it has 320 lines of C code and 702 lines of CN specification.

The allocator follows a *buddy* scheme: it subdivides its available memory into units of 2^n pages for a range of values of n, called *orders*. It maintains a free list of groups of pages for each such size. A client can request a contiguous aligned block of pages at such an order, for which the allocator will look in the corresponding free list. If that is empty, it will look in successively larger free lists, splitting a larger block as needed. Conversely, when a client releases a block (actually, when it decrements a reference count to zero), the allocator will maximally coalesce that block with its sibling, if that is also free, and iterate this upwards.

Allocator memory is divided into contiguous memory regions, called *pools*, each tracked in a `struct hyp_pool` that has the free lists (the actual bodies of the free lists are nodes in free pages).

```
1 struct hyp_pool {
2     struct list_head free_area[MAX_ORDER];
3     phys_addr_t range_start;
4     phys_addr_t range_end;
5     u8 max_order;
6 };
```

Following initialisation using `hyp_pool_init` the allocator owns a large contiguous region of memory available for allocations.

```
1 int hyp_pool_init(struct hyp_pool *pool, u64 pfn, unsigned int nr_pages, unsigned int reserved_pages);
```

Fig. 2 shows the main allocator invariant, the CN `Hyp_pool` predicate. We will not explain this in detail, but note that it involves several iterated separating conjunctions and auxiliary CN predicates, along with intricate numerical detail.

To exercise Fulminate we experimented with injecting errors into the buddy allocator specification or implementation.

For example, we break the `Hyp_pool` predicate with an incorrect page-ownership invariant, omitting one of the conditions in the guard, which requires page ownership only when the `vmemmap` metadata indicates the page is not subsumed by a larger surrounding page group:

```
1 take APs = each(u64 i; (start_i <= i) && (i < end_i)
2                  && ((V[i]).refcount == 0u16)
3                  //    && ((V[i]).order != (hyp_no_order ()))
4                  && ((not (excluded (ex, i)))))
5                 {AllocatorPage(array_shift<PAGE_SIZE_t>(ptr_phys_0, i), true, (V[i]).order)};
```

Commenting out the corresponding line, as above, leads Fulminate to detecting a runtime ownership assertion failure. We run the allocator initialised for a pool of 8 order-0 pages (of 2^{12}-bytes each), under a client that first allocates 2 0-order pages, then hands both back to the allocator (henceforth "2+2 test"); this produces, in 0.37s runtime after 29s generation time, the Fulminate error shown in Fig. 3. In detail, when function `__hyp_attach_page` (output line 2) calls `page_add_to_list_pool` (l4), the `ZeroPage` precondition (l6) fails; `ZeroPage` (l9) requires ownership of a page of zero'ed `Bytes` (l11), in the form of an iterated separating conjunction of `ByteV` resources; however, ownership of these (l14) has already been used up – moved to a higher function call stack depth (l19,20) – presumably due the incorrectly strengthened `Hyp_pool` invariant.

```
predicate {
    struct hyp_pool pool
    , map <u64, struct hyp_page> vmemmap
    , map <u64, struct list_head> APs
}
Hyp_pool (pointer pool_l, pointer vmemmap_l, pointer virt_ptr, i64 physvirt_offset)
{
  let ex = exclude_none ();
  take P = Owned<struct hyp_pool>(pool_l);
  let start_i = P.range_start / page_size();
  let end_i = P.range_end / page_size();
  take V = each(u64 i; (start_i <= i) && (i < end_i))
             {Owned(array_shift<struct hyp_page>(vmemmap_l, i))};
  assert (hyp_pool_wf (pool_l, P, vmemmap_l, physvirt_offset));
  let ptr_phys_0 = cn__hyp_va(virt_ptr, physvirt_offset, 0u64);
  take APs = each(u64 i; (start_i <= i) && (i < end_i)
                  && ((V[i]).refcount == 0u16)
                  && ((V[i]).order != (hyp_no_order ()))
                  && ((not (excluded (ex, i)))))
                 {AllocatorPage(array_shift<PAGE_SIZE_t>(ptr_phys_0, i), true, (V[i]).order)};
  assert (each (u64 i; (start_i <= i) && (i < end_i))
    {vmemmap_wf (i, V, pool_l, P)});
  assert (each (u64 i; (start_i <= i) && (i < end_i)
            && ((V[i]).refcount == 0u16)
            && ((V[i]).order != (hyp_no_order ()))
            && ((not (excluded (ex, i)))))
    {vmemmap_l_wf (i, physvirt_offset, virt_ptr, V, APs, pool_l, P, ex)});
  assert (each(u8 i; 0u8 <= i && i < P.max_order)
             {freeArea_cell_wf (i, physvirt_offset, virt_ptr, V, APs, pool_l, P, ex)});
  return {pool: P, vmemmap: V, APs: APs};
}
```

Fig. 2. Main buddy allocator invariant

We injected another subtle specification bug, into the postcondition of `__hyp_extract_page`, a function involved in handing out pages. When a page at order n is requested, the allocator finds the minimal order $m \geq n$ for which a free page p is available (if any). It calls `__hyp_extract_page` to extract p, breaking apart larger pages that may subsume p and removing any free-list entries, and calls `hyp_set_page_refcounted(p)`. Among various wellformedness conditions, the `Hyp_pool` invariant requires that pages not subsumed by larger pages, and with 0 refcount, must be part of a free list. Following `__hyp_extract_page` and before `hyp_set_page_refcounted(p)`, p is not, and so a correct postcondition of `__hyp_extract_page` has to carefully "carve out" p as an exception to the `Hyp_pool` invariant at that point. The same 2+2 test from above catches an incorrect `__hyp_extract_page` postcondition lacking this exception, in 1.40s (28s with generation and compilation).

We also experimented with breaking the allocator code. We modified `__hyp_attach_page`, a central function responsible for re-attaching pages returned to the allocator, to incorrectly coalesce pages (omitting a `p = min(p, buddy)`); using the 2+2 test from above, in 0.37s runtime (and 31s generation time) Fulminate flags an error, indicating that a function call by `__hyp_attach_page` violates an alignment precondition of `__find_buddy_avail`.

```
...
function __hyp_attach_page, file driver-exec.c, line 3350
************************************************************
function page_add_to_list_pool, file driver-exec.c, line 2740
original source location:
/*@ requires take ZP = ZeroPage(virt, true, order); @*/
                  ^cn-pKVM-buddy-allocator-case-study//driver.pp.c:1297:19:
************************************************************
function ZeroPage, file cn.c, line 658
original source location:
    take Bytes = each (u64 i; (vbaseI <= i) && (i < (vbaseI + length)))
         ^cn-pKVM-buddy-allocator-case-study//driver.pp.c:715:10:
************************************************************
function ByteV, file cn.c, line 607
original source location:
  take B = Owned<char>(virt);
       ^cn-pKVM-buddy-allocator-case-study//driver.pp.c:688:8:
Precondition ownership check failed.
  ==> 0x7fc24f801000[0] (0x7fc24f801000) not owned at expected function call stack depth 4
  ==> (owned at stack depth: 5)
```

Fig. 3. Fulminate output for a bug injected into the buddy allocator specification.

Set of examples	**Gen. time**		**Runtime (U)**		**Space (U)**		**Runtime Δ**		**Space Δ**	
	mean	SD	mean	SD	mean	SD	mean	SD	mean	SD
CN tutorial	0.18	0.11	0.22	0.03	520.43	3.94	0.10	0.54	25.70	55.32
Verifast examples	0.29	0.24	0.21	0.02	520.00	0.00	0.02	0.05	31.20	17.30
Buddy (single run)	28.64	–	0.20	–	536.00	–	0.10	–	24064.00	–

Fig. 4. Performance metrics for various benchmark examples. This shows the mean and standard deviation (SD) of the time taken to generate executable CN specifications for an example; the runtime (s) and space usage (KB) of running the uninstrumented code, denoted by **(U)**; the difference in time (s) between running the executable for instrumented code versus uninstrumented code; and the difference in space (KB) between running the executable for these.

5.4 Performance

In §2.2 we hypothesised that for the small critical-code examples that are our main current target, even a relatively naive implementation of Fulminate is fast enough to be useful. We confirm this by measuring its time and space costs for the above examples, looking at the runtime cost of generating the instrumented code, the compilation and linking time of that, and the runtime cost of executing the instrumented code (and the latter compared to the runtime cost of executing the uninstrumented code). For all of these the total time is small, as shown in Fig. 4, so eminently usable. The differences between the examples are in the noise except that for three of the larger examples generation takes an extra 0.2s – this, and the usable but long generation times for buddy, suggest there is some unfortunate scaling in the generator, which we have not yet investigated; it may just be CN well-formedness checking. For the real-world hypervisor code of the buddy allocator, finding the synthetic bugs of §5.3 are likewise quick, as noted above: 0.4s to 1.4s runtime, after around 30s generation time.

Our focus on small critical examples that one aims to verify makes this an interestingly different regime to those of some previous work on dynamic assertion checking, where performance on larger examples has been one of the main goals, as we return to in §6. We hypothesise that the fact that CN assertions can be highly discriminating means that bugs (in code or in specs) can often

# elements	# dynamic `Owned` calls	Runtime (U)	Space (U)	Runtime Δ	Space Δ
2	38	0.22	520	-0.01	64
4	94	0.24	520	-0.02	84
8	278	0.21	520	0.04	144
16	934	0.21	520	0.00	360
32	3398	0.20	520	0.01	1100
64	12934	0.20	520	0.04	3848
128	50438	0.22	520	0.02	14284
256	199174	0.21	520	0.17	55184
512	791558	0.20	520	0.65	216828
1024	3155974	0.20	524	2.70	859596

Fig. 5. Performance metrics for a stack example taken from Software Foundations, Vol. 6 [11], exercised using different numbers of stack elements (from 2^1 to 2^{10}).

# pages	# dynamic `Owned` calls	Runtime (U)	Space (U)	Runtime Δ	Space Δ
2	335613	0.20	536	0.10	24060
4	1096694	0.20	544	0.32	76648
8	4849348	0.22	572	1.35	331252

Fig. 6. Performance metrics for the buddy allocator, exercised using different numbers of pages (2, 4 and 8).

be found with rather small test cases. That is true for the buddy allocator example above, where the test case comprises only four simple function calls into the allocator, though this does involve a nontrivial amount of memory (2^{12} bytes per page + hypervisor metadata) whose ownership Fulminate tracks byte-wise.

Nonetheless, performance on larger examples and test cases will certainly be a concern in future, and there are many obvious optimisations one could do. Here we establish a baseline by measuring the current cost of scalable versions of two of the above: a stack example taken from Software Foundations, Vol. 6 [11], with stacks of increasing size (Fig. 5), and the buddy allocator, with an increasing number of pages to be allocated (Fig. 6). One would expect Fulminate costs to scale polynomially beyond that of the uninstrumented code, as the code is doing traversals of datastructures of increasing size, and in each of those, the CN instrumentation is also doing traversals to mark or check ownership and compute the CN predicates that abstract from the concrete heap state. The space usage here essentially measures the total instrumentation allocation, as we have deferred implementing memory management – because even the very naive approach we currently take, of simply never de-allocating instrumentation allocations, still works for our primary examples. We envisage more sophisticated schemes in due course, partly region-based; they should support concurrency and run bare-metal as hypervisor code. A more informative measure of the Fulminate cost is perhaps the overhead per dynamic occurrence of **`Owned`**, which in the largest examples above is 0.3μs or 0.9μs.

5.5 Dynamic and Static Checking

With Fulminate, CN specifications serve a dual role, both as assertions for dynamic checking, and as specifications for static program proof. We envisage differing interplay between the two in different scenarios.

In some cases, CN assertions may be used solely as generalised assertions, in a rich specification language that can express ownership properties, without any intention to use them as proof. In those cases, the Fulminate ability to dynamically check them will be the only way to check whether the programmers' code and intent (as expressed in such assertions) are consistent or not.

In other cases, where one does aim ultimately as proof, dynamic checking will be useful along the way. Classically, most verified code is developed in concert with its correctness proof, and such

programs are carefully structured to maintain program invariants and ease the proof. However, most code "in the wild" is not developed with proof in mind, and programmers often write code which tangles together multiple invariants, re-establishes invariants just as they are needed, and use many idioms (such as implementing *ad hoc* objects with a record of function pointers) which can require very sophisticated program reasoning techniques to verify. So to verify pre-existing code, one has to do a lot of work to figure out what the invariants actually are – and being able to quickly and easily dynamically check candidate invariants should be a powerful technique for discovering the invariants needed to do verification.

Although anecdotal, our experience in using Fulminate supports this intuition. Two users of CN had attempted to verify a VST example (a string-to-integer hashtable) in CN, but abandoned the effort because of abstract error messages. Using Fulminate, it became possible to find and fix high-level defintions early and often: a few minutes to correct a misspecified hash function; around ninety minutes to correct a double owned in the central hashtable predicate, mostly spent in locating where the ownership was first claimed. This demonstrates an alternative, and arguably more natural way of *refining* simple definitions and specifications across a wide range of functions, rather than guessing all the information needed to verify one function at a time.

6 Related Work

As mentioned in the introduction, there is relatively little work on dynamic testing of separation-logic specifications. Agten et al. [2] do runtime checking of separation-logic contracts for module boundaries. Similar to Fulminate, this is by translating them into C, though relying on a mode distinction rather than the syntactic restriction of CN specifications used here. An integrity check ensures that the untrusted context has not mutated the module state, on entry. Focussing on those module boundaries, this is not (unlike Fulminate) doing fine-grain checks of all separation-logic assertions and of all accesses. Nguyen et al. [37] do runtime checking of separation-logic properties for Java, recording ownership in marks associated with each object and checking and adjusting it in pre- and post-conditions. The presence of disjunction in their specification language requires undoing this colouring when disjuncts fail. It too relies on a mode analysis. Perry et al. [38] and Jia [20] do runtime checking of contracts in a linear logic, adjoining the LolliMon linear logic-programming runtime [27] to an OCaml interpreter for a small C-like language. Calcagno et al. [9] and Brotherston et al. [7] give complexity results and, in the latter, a model-checking algorithm for separation logic. They note that while this is EXPTIME-complete in general, imposing natural syntactic restrictions makes it NP-complete or polynomial. These restrictions essentially rule out various forms of nondeterminism in the model-checking problem, and partly inspired the restrictions that CN assertions impose.

The Dryad logic of Madhusudan et al. [29] introduced a variant of separation logic for heap verification, and like CN (and many other tools), made the observation that working with determined predicates (where the heap splits are determined by the structure of the assertions) simplifies automated verification. This restriction is similar to CN's restriction to precise predicates, though our observation that this makes runtime checking of separation logic predicates feasible appears to be novel.

At present every time CN checks an assertion it traces the heap anew, but the Ditto work of Shankar and Bodík [50] exploits the observation that heap assignments happen one-by-one to give an incremental algorithm for checking that data structure invariants are maintained. Gyori et al. [16] build on this work to give specialized (and optimized) algorithms for the case of list-manipulating programs. These papers suggest that it is potentially possible to incrementalise CN heap-shape checking, which could significantly decrease the cost of runtime checking.

The Stacked Borrows operational semantics for Rust [21] dynamically checks a borrowing discipline on pointers. It tracks the flow of ownership between pointer aliases by recording, per memory location, a stack of borrowing pointers and their permission types (rather than the function call stack depth at which the location is owned, as in Fulminate), and checks pointer accesses and creation. Its main purpose is for analysing compiler optimisations, but it has also been implemented as an executable checker to test aliasing disciplines in unsafe Rust code. While Fulminate's C translation allows ownership testing within the existing language and using standard debugging tools, Stacked Borrows relies on the Miri Rust interpreter.

Implicit Dynamic Frames [54] is a variant of separation logic that supports executable heap-dependent expressions in assertions. Smans et al. discuss the possibility of runtime checking of specifications as one advantage, and draw parallels between Nguyen et al. [37]'s implementation of separation logic runtime checking and the access sets used by Implicit Dynamic Frames. The paper, however, focuses on proof, not testing.

There is a large literature on non-separation-logic runtime testing and its interactions with verification, dating back to Euclid [24] and Anna [28] in the 1970s, with Eiffel's design by contract being particularly influential. Hatcliff et al. [17] give a useful survey of behavioural interface specification languages. Leavens et al. [26] describe the JML integration of runtime checking and verification. Frama-C's E-ACSL [6, 15, 39, 51–53] is an important example of these. Developed over the last decade, it translates the first-order logic specifications of ACSL into executable C code, and it also supports some custom memory footprint predicates, with optimised runtime ghost state to support that. It is substantially more engineered than Fulminate, but (inheriting from ACSL) is not designed for a separation-logic view of ownership. For embedded software, ownership may not be a primary concern, but for the critical systems software we target, it certainly is.

Several papers discuss explaining proof failures with some form of runtime tests: Müller and Ruskiewicz [35], Petiot et al. [40], and Kosmatov et al. [23]. Christakis et al. [12] describe Delfy, a system for exploring Dafny verification failures with testing.

AddressSanitizer and MemorySanitizer [47, 55] are now widely used and highly optimised, but check only the implicit specifications of the absence of certain C undefined behaviours.

Further from testing, separation-logic static analysis has been very successful in Infer [1, 8], and more recently incorrectness separation logic is also a promising bug-finding technique [25]; both are much more elaborate than concrete testing, and aimed at implicit specifications and large code bases rather than the explicit specifications for small critical code that we target here.

7 Conclusion: Limitations and Future Work

Fulminate covers most of the CN specification language and much of C. As we have seen, this is enough to handle substantial small examples, and for the small critical code that it targets (in domains where users aim to write specifications), that is already interesting. However, there remain several features of CN and C to be supported, that will be required for a production-quality tool. Many of these are straightforward engineering, with a few more substantial.

On the CN side, Fulminate currently assumes that CN maps are indexed by bitvectors or integers rather than the arbitrary CN base types supported by the proof tool. This will require generating comparison functions for CN types used as map indices, much like we currently generate equality functions for all used CN types. CN distinguishes uninitialised **`Block`** and initialised **`Owned`** resources, but Fulminate does not. This will need an additional bit in the ghost state for each allocated memory byte.

On the C side, Fulminate currently does not support **`goto`** into or out of blocks. The infrastructure to compute the corresponding ownership ghost-state updates is in place; it just needs to be plumbed in. Like CN, it currently does not support multi-compilation-unit programs (though it does support

simple `#includes`), or C unions. Like Cerberus and CN, it operates on C source after preprocessor macro expansion, which is semantically correct but can be annoying when writing specifications or reporting errors.

Our focus to date has been on writing a correct checker, not a fast checker (beyond the basic architectural choice to do on-line checking in readable generated C). CN and Fulminate target small critical code, for which checker performance is less of a concern than in some other domains, and it is more than fast enough for the examples we described, but there are nonetheless many potential optimisations worth doing. Most acutely, it will need a bare-metal concurrent garbage collector that can run at EL2.

Fulminate checks user-written specifications. The Cerberus elaboration from Typed Ail to Core identifies the implicit specification of all the C sequential undefined behaviours, but because Fulminate operates as a source-to-source translation from the Typed Ail, that information is not readily available. It is unclear whether the industry sanitisers already practically identify all those cases, or whether tooling based on the Cerberus Core execution, or an extension of Fulminate that adds explicit such checks (as Frama-C can), would be worthwhile.

Fulminate is structured as an annotated-C source to C source translation, so that the result can be built and executed conventionally. A quite different design would be to directly execute concrete operational semantics for the programming and specification languages, but with the same runtime tracking of ownership. That would have substantial pros and cons that would be interesting to explore, both for CN and for other verification tools.

CN, and thus Fulminate, do not yet support concurrency, though their separation-logic foundation should make sequentially consistent and release-acquire concurrency relatively straightforward to add. In the instrumentation, this could just record the owning hardware thread ID, and separation-logic permissions, along with the existing stack depth. The ownership of lock invariants could be done simply with "fake" stack depths.

Beyond executable checking it would be very interesting to see if there are other ways of exploiting Fulminate's code generation: since Fulminate outputs regular C code it may be possible to use it in conjunction with property-based testing, or to guide fuzzing, or with existing C analysis tools, such as using CBMC for model checking against separation logic specifications.

Finally, we look forward to evaluating how Fulminate is received by non-academic users, in planned HCI studies. Assessing and tuning the user experience will be a very interesting direction following the current work.

Acknowledgments

We thank Benjamin Pierce, Mike Dodds, Cole Schlesinger, Zain Aamer, and other members of the VERSE project for discussions and for their early experiments with Fulminate. We thank Ben Laurie and Sarah de Haas (Google) for their support.

This work was funded in part by Google. This work was funded in part by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee for ERC-AdG-2022, EP/Y035976/1 SAFER. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 789108, ERC-AdG-2017 ELVER). The authors would like to thank the Isaac Newton Institute for Mathematical Sciences, Cambridge, for support and hospitality during the programme Big Specification, where work on this paper was undertaken. This work was supported by EPSRC grant EP/Z000580/1.

References

[1] 2024. Infer. https://fbinfer.com/. Accessed 2024-07-08.

[2] Pieter Agten, Bart Jacobs, and Frank Piessens. 2015. Sound Modular Verification of C Code Executing in an Unverified Context. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 581–594. https://doi.org/10.1145/2676726.2676972

[3] Andrew W. Appel, Lennart Beringer, and Qinxiang Cao. 2023. *Verifiable C*. Software Foundations, Vol. 5. Electronic textbook. https://softwarefoundations.cis.upenn.edu Version 1.2.2.

[4] Vytautas Astrauskas, Aurel Bílý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13260)*, Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer, 88–108. https://doi.org/10.1007/978-3-031-06773-0_5

[5] Rini Banerjee, Kayvan Memarian, Dhruv Makwana, Christopher Pulte, Neel Krishnaswami, and Peter Sewell. 2024. Supplementary material for Fulminate: Testing CN Separation-Logic Specifications in C. Online, http://www.cl.cam.ac.uk/users/pes20/cn-testing-popl2025.pdf, accessed 2024-11-23.

[6] Thibaut Benjamin and Julien Signoles. 2023. Abstract Interpretation of Recursive Logic Definitions for Efficient Runtime Assertion Checking. In *Tests and Proofs - 17th International Conference, TAP 2023, Leicester, UK, July 18-19, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14066)*, Virgile Prevosto and Cristina Seceleanu (Eds.). Springer, 168–186. https://doi.org/10.1007/978-3-031-38828-6_10

[7] James Brotherston, Nikos Gorogiannis, Max I. Kanovich, and Reuben Rowe. 2016. Model checking for symbolic-heap separation logic with inductive predicates. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 84–96. https://doi.org/10.1145/2837614.2837621

[8] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33

[9] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. 2001. Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science, 21st Conference, Bangalore, India, December 13-15, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2245)*, Ramesh Hariharan, Madhavan Mukund, and V. Vinay (Eds.). Springer, 108–119. https://doi.org/10.1007/3-540-45294-X_10

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

[11] Arthur Charguéraud. 2024. *Separation Logic Foundations*. Software Foundations, Vol. 6. Electronic textbook. https://softwarefoundations.cis.upenn.edu Version 2.2.

[12] Maria Christakis, K. Rustan M. Leino, Peter Müller, and Valentin Wüstholz. 2016. Integrated Environment for Diagnosing Verification Errors. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9636)*, Marsha Chechik and Jean-François Raskin (Eds.). Springer, 424–441. https://doi.org/10.1007/978-3-662-49674-9_25

[13] Will Deacon. 2020. Virtualisation for the Masses: Exposing KVM on Android. KVM Forum slides, https://mirrors.edge.kernel.org/pub/linux/kernel/people/will/slides/kvmforum-2020-edited.pdf. Accessed 2022-07-07.

[14] Will Deacon. 2020. Virtualization for the Masses: Exposing KVM on Android. https://www.youtube.com/watch?v=wY-u6n75iXc. KVM Forum Talk.

[15] Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. 2013. Common specification language for static and dynamic analysis of C programs. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, Sung Y. Shin and José Carlos Maldonado (Eds.). ACM, 1230–1235. https://doi.org/10.1145/2480362.2480593

[16] Alex Gyori, Pranav Garg, Edgar Pek, and P. Madhusudan. 2017. Efficient Incrementalized Runtime Checking of Linear Measures on Lists. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 310–320. https://doi.org/10.1109/ICST.2017.35

[17] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. 2012. Behavioral interface specification languages. *ACM Comput. Surv.* 44, 3 (2012), 16:1–16:58. https://doi.org/10.1145/2187671.2187678

[18] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

[19] Bart Jacobs, Jan Smans, and Frank Piessens. 2024. The VeriFast Program Verifier: A Tutorial. https://doi.org/10.5281/zenodo.13380705

[20] Limin Jia. 2008. *Linear Logic and Imperative Programming.* Ph. D. Dissertation. Princeton.

[21] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL (2020), 41:1–41:32. https://doi.org/10.1145/3371109

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

[23] Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. 2016. Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9952)*, Tiziana Margaria and Bernhard Steffen (Eds.). 461–478. https://doi.org/10.1007/978-3-319-47166-2_32

[24] Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek. 1977. Report on the programming language Euclid. *ACM SIGPLAN Notices* 12, 2 (1977), 1–79. https://doi.org/10.1145/954666.971189

[25] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–27. https://doi.org/10.1145/3527325

[26] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. 2002. How the Design of JML Accomodates Both Runtime Assertion Checking and Formal Verification. In *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures (Lecture Notes in Computer Science, Vol. 2852)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 262–284. https://doi.org/10.1007/978-3-540-39656-7_11

[27] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. 2005. Monadic concurrent linear logic programming. In *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*, Pedro Barahona and Amy P. Felty (Eds.). ACM, 35–46. https://doi.org/10.1145/1069774.1069778

[28] David C. Luckham and Friedrich W. von Henke. 1985. An Overview of Anna, a Specification Language for Ada. *IEEE Softw.* 2, 2 (1985), 9–22. https://doi.org/10.1109/MS.1985.230345

[29] Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. 2012. Recursive proofs for inductive tree data-structures. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 123–136. https://doi.org/10.1145/2103656.2103673

[30] Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 827–850. https://doi.org/10.1007/978-3-030-81688-9_38

[31] Luc Maranget. 2008. Compiling pattern matching to good decision trees. In *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, Eijiro Sumii (Ed.). ACM, 35–46. https://doi.org/10.1145/1411304.1411311

[32] Kayvan Memarian. 2023. *The Cerberus C semantics.* Technical Report UCAM-CL-TR-981. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-981 PhD thesis.

[33] 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. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3290380 Proc. ACM Program. Lang. 3, POPL, Article 67. Also available as ISO/IEC JTC1/SC22/WG14 N2311.

[34] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 1–15. https://doi.org/10.1145/2908080.2908081

[35] Peter Müller and Joseph N. Ruskiewicz. 2011. Using Debuggers to Understand Failed Verification Attempts. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6664)*, Michael J. Butler and Wolfram Schulte (Eds.). Springer, 73–87.

https://doi.org/10.1007/978-3-642-21437-0_8

[36] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2017. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Dependable Software Systems Engineering*, Alexander Pretschner, Doron Peled, and Thomas Hutzelmann (Eds.). NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 50. IOS Press, 104–125. https://doi.org/10.3233/978-1-61499-810-5-104

[37] Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. 2008. Runtime Checking for Separation Logic. In *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 4905)*, Francesco Logozzo, Doron A. Peled, and Lenore D. Zuck (Eds.). Springer, 203–217. https://doi.org/10.1007/978-3-540-78163-9_19

[38] Frances Perry, Limin Jia, and David Walker. 2006. Expressing heap-shape contracts in linear logic. In *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*, Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen (Eds.). ACM, 101–110. https://doi.org/10.1145/1173706.1173723

[39] Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov, and Julien Signoles. 2014. Instrumentation of Annotated C Programs for Test Generation. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*. IEEE Computer Society, 105–114. https://doi.org/10.1109/SCAM.2014.19

[40] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. 2016. Your Proof Fails? Testing Helps to Find the Reason. In *Tests and Proofs - 10th International Conference, TAP@STAF 2016, Vienna, Austria, July 5-7, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9762)*, Bernhard K. Aichernig and Carlo A. Furia (Eds.). Springer, 130–150. https://doi.org/10.1007/978-3-319-41135-4_8

[41] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying systems C code with separation-logic refinement types. In *Proceedings of the 50th ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3571194

[42] Christopher Pulte, Benjamin C. Pierce, Cole Schlesinger, and Elizabeth Austell. 2024. CN tutorial. https://rems-project.github.io/cn-tutorial/. [Online; accessed 26-October-2024].

[43] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[44] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. https://doi.org/10.1145/1375581.1375602

[45] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 158–174. https://doi.org/10.1145/3453483.3454036

[46] José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, part i: a multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 927–942. https://doi.org/10.1145/3385412.3386014

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[48] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 77–87. https://doi.org/10.1145/2737924.2737964

[49] Amir Shaikhha, Andrew W. Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing, FHPC@ICFP 2017, Oxford, UK, September 7, 2017*, Phil Trinder and Cosmin E. Oancea (Eds.). ACM, 12–23. https://doi.org/10.1145/3122948.3122949

[50] Ajeet Shankar and Rastislav Bodík. 2007. DITTO: automatic incrementalization of data structure invariant checks (in Java). In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 310–319. https://doi.org/10.1145/1250734.1250770

[51] Julien Signoles. 2018. *From Static Analysis to Runtime Verification with Frama-C and E-ACSL*. https://tel.archives-ouvertes.fr/tel-04469397

[52] Julien Signoles. 2021. The e-ACSL perspective on runtime assertion checking. In *VORTEX 2021: Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution, Virtual Event, Denmark, 12 July 2021*, Wolfgang Ahrendt, Davide Ancona, and Adrian Francalanza (Eds.). ACM, 8–12. https://doi.org/10.1145/3464974.3468451

[53] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. 2017. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper). In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA (Kalpa Publications in Computing, Vol. 3)*, Giles Reger and Klaus Havelund (Eds.). EasyChair, 164–173. https://doi.org/10.29007/FPDH

[54] Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit Dynamic Frames. *ACM Trans. Program. Lang. Syst.* 34, 1 (2012), 2:1–2:58. https://doi.org/10.1145/2160910.2160911

[55] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, Kunle Olukotun, Aaron Smith, Robert Hundt, and Jason Mars (Eds.). IEEE Computer Society, 46–55. https://doi.org/10.1109/CGO.2015.7054186

[56] Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. 2024. VST-A: A Foundationally Sound Annotation Verifier. *Proc. ACM Program. Lang.* 8, POPL (2024), 2069–2098. https://doi.org/10.1145/3632911

Received 2024-07-11; accepted 2024-11-07