CN: Verifying Systems C Code with Separation-Logic Refinement Types

CHRISTOPHER PULTE, University of Cambridge, UK
DHRUV C. MAKWANA, University of Cambridge, UK
THOMAS SEWELL, University of Cambridge, UK
KAYVAN MEMARIAN, University of Cambridge, UK
PETER SEWELL, University of Cambridge, UK
NEEL KRISHNASWAMI, University of Cambridge, UK

Despite significant progress in the verification of hypervisors, operating systems, and compilers, and in verification tooling, there exists a wide gap between the approaches used in verification projects and conventional development of systems software. We see two main challenges in bringing these closer together: verification handling the complexity of code and semantics of conventional systems software, and verification usability.

We describe an experiment in verification tool design aimed at addressing some aspects of both: we design and implement CN, a separation-logic refinement type system for C systems software, aimed at predictable proof automation, based on a realistic semantics of ISO C. CN reduces refinement typing to decidable propositional logic reasoning, uses first-class resources to support pointer aliasing and pointer arithmetic, features resource inference for iterated separating conjunction, and uses a novel syntactic restriction of ghost variables in specifications to guarantee their successful inference. We implement CN and formalise key aspects of the type system, including a soundness proof of type checking. To demonstrate the usability of CN we use it to verify a substantial component of Google's pKVM hypervisor for Android.

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

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

ACM Reference Format:
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. *Proc. ACM Program. Lang.* 7, POPL, Article 1 (January 2023), 32 pages. https://doi.org/10.1145/3571194

1 INTRODUCTION

Systems software, such as hypervisors, operating systems, and compilers, is critical infrastructure: errors can compromise the correctness and security of all software running above it. This motivates significant effort into ensuring that it works as intended, and recent years have seen major progress in verified hypervisors [Baumann et al. 2016; Guanciale et al. 2016; Heiser et al. 2020; Klein et al. 2014; Leinenbach and Santen 2009; Li et al. 2021; Tao et al. 2021], operating systems [Gu et al. 2016], and compilers [Amadio et al. 2013; Fox et al. 2017; Kumar et al. 2014; Leroy 2009; Tan et al. 2016],

Authors' addresses: Christopher Pulte, University of Cambridge, UK, Christopher.Pulte@cl.cam.ac.uk; Dhruv C. Makwana, University of Cambridge, UK, Dhruv.Makwana@cl.cam.ac.uk; Thomas Sewell, University of Cambridge, UK, Thomas.Sewell@cl.cam.ac.uk; Kayvan Memarian, University of Cambridge, UK, Kayvan.Memarian@cl.cam.ac.uk; Peter Sewell, University of Cambridge, UK, Peter.Sewell@cl.cam.ac.uk; Neel Krishnaswami, University of Cambridge, UK, Neel.Krishnaswami@cl.cam.ac.uk.

2475-1421/2023/1-ART1
https://doi.org/10.1145/3571194

and in verification tooling for low-level code, e.g. [Astrauskas et al. 2022; Barnett et al. 2005; Baudin et al. 2021; Cao et al. 2018; Hawblitzel et al. 2014; Jacobs et al. 2011; Lepigre et al. 2022; Malecha et al. 2022; O'Connor et al. 2021; Sammler et al. 2022, 2021; Swamy et al. 2016].

However, there is a wide gulf between the approaches used for this and the world of conventional systems software development. In most verified software projects, the system was designed and written from the outset with verification in mind (indeed, standard wisdom has been that to do otherwise would be foolhardy), and the languages, tools, and skills used are far from those of conventional systems developers, e.g. Coq proof v.s. C programming. This divergence has been necessary to make progress, and in some contexts is perfectly acceptable for practical use (where the verified software can be developed and maintained by an expert verification team, and is not intertwined with a larger conventional development) – but it significantly impedes the broader adoption of verification, which remains very costly to do and maintain. In an ideal world, we would be able to verify the code written by conventional development teams, to do so with a minimum of highly specialised expertise, and to maintain such verifications at reasonable cost. We see two main challenges in more closely approaching such an ideal world.

Handling conventional systems software. Conventional systems software relies on many low-level idioms: pointer arithmetic and manipulation of pointer representations, custom memory allocators, complex ownership patterns, function pointers, and so on. Much is written in C, to enable those idioms, to exploit the long-established infrastructure of C compilers and linkers, and to use the long-established systems developer community skills. This C is not the idealised imperative language of some programming language research, but the "real thing", and it has a particularly complicated semantics, with undefined behaviours, complex control-flow, implicit type coercions, and subtle arithmetic semantics. Many restrictions that one might like to make to ease verification, e.g. to forbid manipulation of pointer representations, passing the address of a local variable to a callee, or complex ownership disciplines, would not be acceptable for conventional production systems code. Systems software also has to manage systems aspects of the underlying architecture that are not expressible at the C abstraction, including address translation, instruction-cache, data-cache, and TLB management, and exceptions, and the relaxed concurrent behaviour of all these.

Verification usability. In the context of conventional software development and deployment, verification usability becomes essential, but rather than focusing simply on whether a one-off verification was *possible* (by a highly expert team), which was the headline result of many early papers, we need to simultaneously minimise verification cost and the required expertise. We also need to support maintenance as a first-class goal: software evolves with time and proofs need to be maintained along with the code. In the limit, one would aim to get to the point where full verification can be done and maintained by the conventional development team themselves. It is unclear whether that will ever be realistic, but we certainly want developers to read and review the assertions used in verification, and (in due course, and with some training) to write them, and hopefully to maintain verifications in the face of modest changes to the code.

Usability brings conflicting requirements between *automation* and *predictability.* Previous work has explored a wide range of approaches, from fully automated SMT techniques through to manual mechanised proof in a proof assistant. Some automation is necessary to reduce cost, proof maintenance effort, and the required expertise, but it can make verification failures unpredictable, and inscrutable for those without deep knowledge of the internals of a tool. To provide a better user experience for verifiers than has normally been available, verification tooling should be reasonably predictable, reliably accepting or rejecting the code, without relying on heuristics that may unpredictably fail in the face of minor code changes. Moreover, verification failures should be explained with diagnosable errors, ideally in the form of counterexamples.

In this paper we describe the design of a verification tool, CN, that reconciles some aspects of both challenges (supporting all the above remains well beyond the state of the art, obviously). It is based on two main choices:

We build on an accurate ISO C semantics. Rather than use an idealised "C-like" language, as assumed by many other tools, or developing a new custom C semantics (either explicit or implicit in the verification tool), as done by others, we use Cerberus [Memarian et al. 2016], a well-validated explicit semantics for a large fragment of ISO C. Cerberus defines C by elaboration into the Core language which our type system targets. This gives us the high coverage of C features needed for verifying conventional production C code, and some confidence in CN's soundness with respect to that semantics. Currently CN's treatment of this semantics has two limitations. First, we do not handle C's weakly sequenced or unsequenced memory actions; we plan to extend CN to support these along the lines of Frumin et al. [2019]. Second, we use a concrete memory object model (pointers are essentially just integers); to correctly handle integer-pointer casts we plan to use the VIP model [Lepigre et al. 2022]. We expect both additions to be straightforward. We diverge from ISO C by not modeling effective types, as much systems code (including our main target) is compiled without them, using '-fno-strict-aliasing', and as the exact ISO intent for them remains unclear.

We develop a separation-logic refinement type system with an SMT backend, designed with careful restrictions to guarantee that inference always succeeds or fails. We see two main aspects to usability. First, we want to perform verification compositionally along the code structure, so that specifications better match developer intuition; so that we can give more localised error reports; and so that verification scales better and can be maintained better. Many different approaches work compositionally: program logics, type systems, some forms of refinement reasoning, compositional symbolic execution methods, and so on. Moreover, many of these techniques are closely related – for example, [Melliès and Zeilberger 2015] show how program logics and refinement types have a common underlying semantics. Our second aspect leads us to a choice among these: we wanted CN to be predictable, with every input program either cleanly accepted or rejected. This leads us to work in terms of a substructural refinement *type system* with linear resource types. Many existing verification tools are designed to work on a best-effort basis, but there is an extensive literature on decidable type inference for refinement types (e.g., liquid types [Rondon et al. 2008]). Combining this with a substructural resource discipline makes it possible to give functions local specifications in terms of their relevant memory footprint, and to cleanly specify those footprints in a value-dependent way – for example, the traditional C linked list is a pointer which points to a block of memory only if it is not null. In addition, programmers are already familiar with type systems, and so there is a pre-existing place in their workflow where we can insert CN.

One motivating verification target for CN is pKVM [Deacon 2020; Edge 2020], a hypervisor developed by Google, intended to be widely deployed on Android phones to ensure isolation between a Linux kernel (untrusted after initialisation) and guest virtual machines. It will be included in Android 13 [Android Open Source 2022, "Android Virtualization Framework"]. pKVM runs as a Type 1 hypervisor distinct from the Android Linux kernel, but it is developed as part of the kernel tree, written in C and Arm assembly, using various Linux kernel header files and other code, following normal Linux kernel development methods, and compiled with Clang. This context is fixed, and practical verification of pKVM — or of similar systems code — has to accommodate it, not redefine the problem with a clean-slate approach.

Contributions

We have designed CN, a separation-logic refinement type system for C. CN has first-class linear resource types that easily support pointer aliasing and computed access via pointer arithmetic, resource inference with built-in knowledge of the layout of C types, and user-defined inductive predicates. To make inference predictable we have introduced a novel syntax for separation logic assertions which uses variable scoping to ensure that programmers write formulae in such a way that inference of ghost variables and existential witnesses always succeeds. Furthermore, the refinement type system is carefully engineered to ensure that it always produces logical constraints which fall into an SMT fragment known to be decidable (quantifier-free formulas using the theories of uninterpreted functions, linear integer arithmetic, records, and extensional arrays). We support properties falling outside this fragment via mechanisms to package entailments into lemmas which can be exported to Coq, and to manually invoke these lemmas and instantiate quantifiers.

We show it is possible to build verification tools for an accurate ISO C semantics. Rather than an idealised "C-like" language, as assumed by many other tools, or a custom embedding of a fragment, as used by others, we use Cerberus [Memarian et al. 2016], a semantics for a large fragment of ISO C. Cerberus defines C by elaboration into its Core language, which our type system targets, and it has been validated on substantial C test suites. This gives us confidence in CN's soundness (albeit without foundational proof) and the high coverage of C features needed for verifying conventional production C code.

We have implemented CN, an open-source tool for C verification. CN elaborates C into Core, type checks the Core, and rejects programs with C undefined behaviour or which fail their specification. CN delegates refinement subtyping and pointer-equality reasoning to the Z3 SMT solver [De Moura and Bjørner 2008]. It is available in the online materials (see below).

We verify the pKVM hypervisor's buddy allocator in CN. As a case study, we verify a substantial component of pKVM: the buddy allocator it uses for managing page-table memory. The code of the buddy allocator was pre-existing code, written by the pKVM developers before we began work on CN. We are able to verify the code largely as written (except for locks and minor changes detailed later), demonstrating that CN handles non-trivial pointer arithmetic and aliasing data structures, difficult quantified well-formedness invariants, and non-linear integer arithmetic.

We formalise key parts of the type system. We prove soundness of type checking (although not inference), increasing confidence in the type system design and implementation.

The online materials, at www.cl.cam.ac.uk/~cp526/popl23.html, contain the CN source, our formalisation, the buddy allocator verification, and a case study comparing CN with other tools.

Limitations. CN is designed to address some important aspects of verification-tool usability, but we discuss and evaluate usability only in ways typical of the verification literature: by describing the verification of a substantial example, done by the tool authors. Ultimately one would like empirical user studies, but that is a separate (and very interesting) research problem in its own right, and will need additional tool development. CN currently does not support recursive specification functions; these are easily added, together with a mechanism for users to manually unfold definitions. We base CN on Cerberus to soundly handle most C features and their complex semantics, but CN does not currently support unions; moreover, we use a concrete memory object model, and we currently ignore the sequencing strengths of memory actions (in C and Cerberus); we plan to incorporate [Lepigre et al. 2022]'s VIP memory object model and strengthen CN so that one can prove that code is not sensitive to C's loose evaluation order. We focus to date only on sequential verification, but plan to extend the CN separation-logic refinement type system with some of the extensive research on separation-logic concurrency. All these should be relatively straightforward extensions.

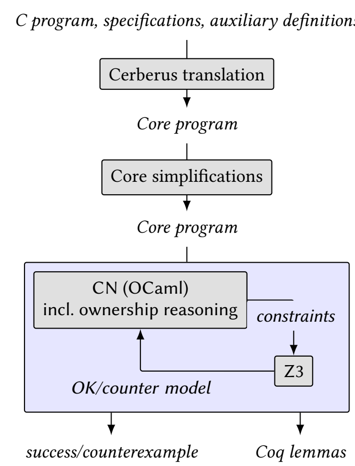
We now explain our type system design and the choices aimed at handling conventional production systems software and verification usability (§2), demonstrate the usability of CN in the verification of pKVM's buddy allocator (§3), explain the formalisation of key type system aspects (§4), compare with the Frama-C, RefinedC, and VeriFast tools (§5), and discuss related work (§6).

2 THE CN DESIGN

We now describe the design of CN, balancing expressivity, automation, and predictability.

2.1 Handling a Realistic C Semantics

For a refinement type system for any programming language to be a good one, it has to soundly and sufficiently accurately capture its semantics. With C this is a significant challenge: C has a notoriously complex semantics, with undefined behaviours (UB), implementation-defined behaviours, unspecified values, implicit type coercions, mutable local variables that can be addressed with pointers, complex control flow and variable scoping, and under-specified sequencing of memory accesses. In designing a type system for C one faces the question of how to handle this complexity.



Working at a level close to the C source is desirable for easily reporting errors in terms of the source program. However, doing this for a realistic C semantics would mean replicating much of that in the typing rules, which would be complex and error-prone. Instead, we use Cerberus [Memarian et al. 2019, 2016], a well-validated semantics for a large fragment of C, defined *by elaboration* into *Core*, a much simpler first-order language with pure values; the elaboration makes the complexities of C explicit in the produced Core programs. We use Cerberus to reduce the problem of verifying a C program to verifying its Core elaboration, which we do by *type checking the Core program.* Since Core's semantics is (mostly) straightforward, designing a refinement type system for Core is much easier than doing so for C directly.

Given an input C file, annotated with CN types for functions and loops, in-function CN instrumentation, and user-defined predicates (both described later), CN translates C to Core using Cerberus. CN type annotations in the source are combined with C types in function declarations, and mapped to CN types for the Core program. This translation from C-source types and instrumentation to Core relies on the fact that *Cerberus' elaboration is compositional* in the structure of the C program: C functions are mapped to Core functions, C loop bodies are mapped to Core goto labels/procedures, and the expressions in the Core program follow the structure of the C statements. This allows CN to map C function types to Core function types, C loop (invariant) types to Core procedure types, and in-function CN instrumentation from the C source to CN Core expressions in the correct position. Finally, CN applies some simplifying Core rewrites and type-checks the resulting Core program.

2.2 Core

Consider the simple C function increment, which takes a `signed int i`, increments `i`, and returns it. The produced Core function is shown below (slightly simplified for presentation). Core maps all C control flow to if-then-else and goto; a Core function comprises a collection of labeled blocks/non-return procedures, here just the distinguished entry procedure body (3), and a return procedure, here `ret1` (2), without a code body. Returning from a function is expressed as calling the return procedure (14). Since C has mutable local variables (including function arguments), Core allocates them explicitly: they are allocated on function entry (4), initialised (5), read and written using memory loads (6,12) and stores (11), and de-allocated on function exit (13). In place of C's fixed-width integers Core uses unbounded

```
signed int increment(signed int i) {
  i = i + 1;
  return i;
}
```

integers; e.g. function argument i (1) has type `integer`= $\mathbb{Z}$; C integer operations are mapped to operations in $\mathbb{Z}$, with explicit handling of unrepresentable values: here line 8's '+' is addition in $\mathbb{Z}$ and line 9 has an undefined behaviour (UB) assertion checking for signed integer overflow. CN proves the absence of UB by proving that all such assertions succeed. Finally, C arithmetic involves implicit type coercions. Core makes these explicit using `conv_int` (8,11,14); for a given target C type τ, `conv_int(`τ`) : integer -> integer` ensures the return value is representable at τ; in the case of an unrepresentable value, it either specifies wrap-around semantics (for unsigned types) or UB (for signed types). Here the `conv_int` calls are no-ops since all arithmetic happens at `signed int` type.

```
1  proc increment (i: integer): integer :=
2    return label ret1
3    body =
4      let i_l: pointer = create(4, 'signed int') in
5      store('signed int', i_l, i);
6      let v1: integer = load('signed int', i_l) in
7      let sv: integer =
8        let n: integer = conv_int('signed int', v1) + 1 in
9        assert_undef(-2147483648 <= n /\ n <= 2147483647, <<UB036>>);
10       n in
11     store('signed int', i_l, conv_int('signed int', sv));
12     let v2: integer = load('signed int', i_l) in
13     kill('signed int', i_l);
14     run ret1(conv_int('signed int', v2))
```

2.3 Refinement Types

We now recall basic refinement types [Rondon et al. 2008] and show how they appear in CN. For the previous `increment` function to behave correctly, we must ensure `i` does not overflow, or `increment` will have undefined behaviour according to the C standard and the Core program (and CN would reject it). Refinement types allow placing constraints on a function's argument and return values. Here we can use them to specify that `increment` requires `i` to be sufficiently small, and that it returns $i + 1$. The CN type for this is (slightly simplified):

$$\Pi i : \mathsf{integer}.\ (i < \mathsf{power}(2, 32) - 1) \Rightarrow \Sigma \mathit{return} : \mathsf{integer}.\ (\mathit{return} = i + 1) \wedge I$$

Here Π binds the computational argument i, i.e. the (single) runtime argument of the C function; $(i < \mathsf{power}(2, 32) - 1) \Rightarrow \ldots$ specifies a *constraint type* for i; Σ binds the computational return value; and $(\mathit{return} = i + 1) \wedge \ldots$ specifies a constraint type fixing its value; finally I is the "empty" return type, corresponding to the empty-heap assertion emp in separation logic). In C, function arguments, such as `i`, are mutable; in this specification, i refers to the initial value of `i`. Like Core, CN uses unbounded integers of type integer = $\mathbb{Z}$ in place of C's bounded integers (such as `signed int`), and handles integer bounds using constraint types. The full CN type of `increment` is as follows:

$$\Pi i : \mathsf{integer}.\ \mathsf{good}\langle\mathsf{signed\ int}\rangle(i) \Rightarrow (i < \mathsf{power}(2, 32) - 1) \Rightarrow$$
$$\Sigma \mathit{return} : \mathsf{integer}.\ \mathsf{good}\langle\mathsf{signed\ int}\rangle(\mathit{return}) \wedge (\mathit{return} = i + 1) \wedge I$$

This captures the integer range constraints using the predicate good; for pointers, good also includes alignment constraints. In the CN implementation, the computational types and the "good" constraint types do not have to be specified explicitly, as they are implied by the C function declaration. Moreover, since CN targets systems developers, we choose a more C-like concrete syntax, shown above.

```
signed int increment(signed int i)
/*@ requires i < power(2,31) - 1 @*/
/*@ ensures return == i + 1 @*/
{ i = i + 1;
  return i; }
```

CN's expression language includes arithmetic and comparison operations, boolean operations and the _?_:_ if-then-else operator, pointer-offsetting for struct members or array indices, pointer/integer casts, struct member access and update, etc. Constraint types are boolean typed expressions, and

CN supports constraints with a single top-level universal quantifier (we may extend this, if needed). Moreover, users can define non-recursive logical functions, as abbreviation for complex expressions, using a C-like syntax; for instance "`function (bool) positive (integer x) { return x>0; }`" defines a function positive : integer $\rightarrow$ bool.

Refinement typing in CN works roughly as follows. The typing context includes a constraint context Φ. For type checking a function, CN initially adds each argument constraint type to Φ; in the above example we initially have $\Phi = \{\mathsf{good}\langle\mathsf{signed\ int}\rangle(i), (i < \mathsf{power}(2, 32) - 1)\}$. Then CN checks the function body. The return types in the typing rules of an expression or statement have constraint types capturing its semantics, which CN accumulates into Φ. Consider, for instance, the expression `i` in the R-value of the assignment in the `increment` function. This translates to the load (6) in Core, reading from the variable `i_l` (holding `i`'s location on the stack). Assuming this value, according to the resource inference (described later), is v_i, the (slightly simplified) return type of the load is $\Pi r : \mathsf{integer}.\ (r = v_i) \wedge I$, specifying a computational return value r, of integer type, and a constraint type equating it to v_i. Type checking is control-flow-sensitive; for instance, for checking some statement `if (e) S1 else S2`, CN adds to Φ the logical constraint c corresponding to `e` when checking `S1`, and $\neg c$ when checking `S2`.

Finally, CN type checks against the function's return type rt: checking whether each constraint type lc in rt holds, given constraint context Φ. Constraints lc can refer to the return value *return* and other values in the specification (e.g. values in resources, as described later). CN delegates constraint reasoning to the Z3 solver [De Moura and Bjørner 2008]. In this example, the return value is the value of `i` on function exit, say v_i', and rt is $\Sigma return : \mathsf{integer}.\ \mathsf{good}\langle\mathsf{signed\ int}\rangle(return) \wedge (return = i + 1) \wedge I$. CN checks that v_i' has type integer, substitutes v_i' for *return*, and checks using Z3 that $\bigwedge \Phi \Rightarrow \mathsf{good}\langle\mathsf{signed\ int}\rangle(v_i')$ and $\bigwedge \Phi \Rightarrow (v_i' = i + 1)$ hold (where i refers to the initial function argument value); i.e. CN checks that both constraints hold given the assumptions in Φ. Here they will, since Φ records the initial assumptions about `i` (from `increment`'s argument constraint types) and the details of this trace through the program. (The refinement reasoning in checking a call of a function `f` is similar: CN checks at the call site whether, in Φ at that point, all argument constraint types of `f`, instantiated to the concrete arguments of the function call, hold.)

2.4 Decidable Refinement Typing

Our refinement type system design follows the liquid types approach [Rondon et al. 2008], and adds mechanisms to reconcile it with the requirements for systems code. We reduce refinement typing to checking decidable propositional logic formulae, for predictable refinement reasoning using SMT solvers: we target SMT formulas known to be in a decidable fragment (quantifier-free formulas using the theories of uninterpreted functions, linear integer arithmetic, records, and extensional arrays), for which SMT solvers should provide reliable (non-`UNKNOWN`) answers. By itself this is at odds with the requirements of verifying realistic systems software, which can rely on complex invariants with quantifiers and non-linear arithmetic, or indeed use non-linear arithmetic in the code. Our design handles this tension between expressivity and predictable automation as follows.

Non-linear arithmetic. CN rejects specifications with non-linear arithmetic: e.g. for two variables x and y, the expression $x * y$ in a type results in a type error (whereas $x * 512$ is accepted). Where this limits verification, users can specify types referring to user-declared uninterpreted functions; e.g.

```
function (integer) my_mul (integer x, integer y)
```

(without a function body) declares `my_mul`, and the expression `my_mul(x,y)` is accepted by CN.

Moreover, some C operations are intrinsically non-linear (e.g. `x*y`). CN's typing rules for certain arithmetic operations (multiplication, division, exponentiation, etc.) distinguish between constant and non-constant arguments: for linear arithmetic their return types include constraints in terms of standard ("interpreted") integer operations, otherwise the constraints use uninterpreted functions; e.g. for C code `x*4096` CN uses '$*$', whereas for `x*y` it uses a built-in uninterpreted function mul_uf, and warns the user about this. Since uninterpreted functions are opaque to CN and Z3, proofs about them require assistance from the user: users have to state the relevant properties in lemmas and explicitly apply these (detailed in §2.11).

Quantifiers. Users can write constraint types with a single universal quantifier. Reasoning about these is reduced to propositional formulae using a simple default quantifier instantiation strategy and support for manual quantifier instantiation by the user. For proving $\bigwedge \Phi \Rightarrow \forall i : bt.\, p[i]$, for some propositional expression p depending on i of base type bt, CN checks if $\forall i : bt.\, \left(\left(\bigwedge \Phi_{\mathrm{qf}} \wedge \bigwedge \Phi_{[i]}\right) \Rightarrow p[i]\right)$ holds, where Φ_{qf} is the quantifier-free subset of Φ and $\Phi_{[i]}$ is the set of all constraints $q[i]$ for which Φ contains $\forall j : bt.\, q[j]$. CN does this by checking that the following quantifier-free SMT formula is unsatisfiable for a fresh constant c: $\left(\bigwedge \Phi_{\mathrm{qf}} \wedge \bigwedge \Phi_{[c]}\right) \wedge \neg p[c]$; i.e. to check whether $\forall i : bt.\, p[i]$ holds in Φ, CN checks that $\neg p[c]$ is impossible assuming (1) all quantifier-free constraints from Φ and (2) all universally quantified constraints from Φ with quantifier base type bt, instantiated at the same constant c. Since this scheme is clearly not complete, users can use lemmas or additionally *manually instantiate quantifiers*: the CN statement "`instantiate` *name*, *e*" instructs CN to instantiate any universally-quantified constraint q in Φ, with the value of expression e, if q mentions predicate *name* and its quantifier has the same base type as e; §3.3 has an example of manual instantiation. Where this is insufficient, lemmas can be used.

Discussion. Past research has investigated decidable fragments of first-order logic with some form of quantification. In particular, Bradley et al. [2006] study the array property fragment, which allows a restricted form of quantified propositions about arrays, sufficiently expressive for specifying properties such as sortedness. We experimented with applying their results, but found the array property fragment insufficiently expressive for encoding properties of arrays required in our setting, in particular certain invariants in the buddy allocator verification.

2.5 Counterexamples

By reducing refinement typing to checking quantifier-free formulae, CN ensures that the resulting constraint problems are SMT-friendly and Z3 gives reliable yes/no answers (not unknown). An important benefit is that when the answer is "no" (i.e. refinement typing has failed), Z3 produces a (trustworthy) counter-model, which CN can use to explain verification failures in terms of concrete values for program variables (which would not be trustworthy in the case of the outcome unknown).

2.6 Resource Types

C programs are stateful: they read, write, allocate and de-allocate memory, and the safety of this has to be verified. For reasoning about memory safety we use resource types based on separation logic. In the example below, `struct s` comprises two `int` members `x` and `y`, and `zero_y` takes a `struct s` pointer and zeroes the pointee's `y` member. To justify the safety of this function, ownership of the memory pointed at by `p` is required.

```
struct s { int x; int y; };
void zero_y (struct s *p) {
    p->y = 0;
}
```

Substructural type systems like Rust [Matsakis and Klock II 2014] closely couple ownership of a memory location to the (computational) types of pointers to that location, to facilitate ownership reasoning: if pointer types and resources are linked, programmers implicitly pass the ownership of the pointers' memory locations, obviating resource inference. CN chooses the opposite approach and uses *first-class resource types* (inspired by L3 [Ahmed et al. 2007]), separating ownership of memory from pointer types. This gives the expressiveness we need, to support pointer aliasing and computed access to data structures via pointer arithmetic (as can be found in C systems code such as the pKVM buddy allocator), at the cost of making type inference more complex. CN uses linear (rather than affine) resource types, so we can (later) encode protocols via resource types.

Returning to the example, we can specify the required memory ownership using CN's Owned resource type. $\mathsf{Owned}\langle\tau\rangle(p)(v)$, for a C-type τ, pointer p and value v, asserts ownership of the memory location pointed at by p and that the value is v. (We slightly modify the presentation of the value v shortly, below.) This is equivalent to a separation logic points-to resource $p \mapsto_\tau v$, indexed by a C-type τ to capture its memory layout [Cao et al. 2018; Krebbers 2016]. The function `zero_y` should receive the ownership of p in the form of an Owned resource argument with some pointee value v, and return an Owned resource for some pointee value v'. We would like to specify this as follows (omitting some details around "good" C values and how v' relates to v):

$$\Pi p : \mathsf{pointer}.\, \forall v : \mathsf{struct\ s}.\, (\mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p)(v)) \twoheadrightarrow\!\!* \\ \Sigma \mathit{return} : \mathsf{unit}.\, \exists v' : \mathsf{struct\ s}.\, (\mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p)(v')) * I$$

Here $r \twoheadrightarrow\!\!* \mathit{ft}$ and $r * \mathit{rt}$ introduce a resource type r into a function type ft or return type rt, respectively, while $\forall$ and $\exists$ are binders for the logical variables v and v'. However, unrestricted use of logical variables via $\forall$ and $\exists$ makes inference infeasible, e.g. by allowing for imprecise specifications (see §2.12). We would like CN to automatically infer the correct instantiation of logical variables (such as v and v' in this example), reliably and without back-tracking.

To this end we impose a simple syntactic restriction that guarantees *inference of logical variables always succeeds*. The restriction is based on partitioning the arguments of a resource into *inputs* and *outputs*, following the intuition that fixing the inputs determines the outputs. For example, in $\mathsf{Owned}\langle\tau\rangle(q)(w)$, CN regards the pointer q as an input and the value w as an output, since for any choice of q at most one value w can exist for which $\mathsf{Owned}\langle\tau\rangle(q)(w)$ holds. We explain the input/output distinction in more detail in §2.12. CN then only allows introducing logical variables for the outputs of resources and replaces general $\forall$ and $\exists$ binders in function and return types with *resource-let-bindings*, which syntactically link the introduction of logical variables to their use as outputs of resources. The type

$$\mathsf{let}\ O = \mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p)$$

specifies ownership of an Owned/points-to resource at type struct s for pointer p and *some* output/value, which it binds to the name O.

Owned has a single output, the pointee value, but user-defined predicates, described in §2.9, can have multiple. In a functional language it would be natural to allow (tuple) pattern-matching in resource-let-bindings, to directly give a name to each output. To make CN syntax as close to C programs as reasonably possible, we instead choose to present resource outputs using records:

- Instead of $\mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p)(v)$, CN specifies the value of an Owned resource using a unary record with field value: $\mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p)\{.\mathsf{value} = v\}$.
- Resource-let-bindings bind a name to a record that is the collection of the resource's outputs; e.g. the name O in $\mathsf{let}\ O = \mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p)$ refers to a unary record with field $O.\mathsf{value}$ for the (single) output of an Owned resource. (In the case of user-defined predicates, O might have multiple fields, one for each output.)

Using resource-let-bindings, the specification we give to `zero_y` (omitting details around good) is:

$$\Pi p : \mathsf{pointer}.\, \mathsf{let}\ O_1 = \mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p) \twoheadrightarrow\!\!* \\ \Sigma \mathit{return} : \mathsf{unit}.\, \mathsf{let}\ O_2 = \mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p) * I$$

Here "$-\!\!*$" is used for resource arguments in function types and "$*$" for resources returned in return types. The scope of O in let $O = R -\!\!* \mathit{ft}$ (resp. let $O = R * \mathit{rt}$) is ft (resp. rt) — i.e. the scoping is the same as with let-bindings if "$-\!\!*$" and "$*$" were replaced with "`in`".

Hence, in the example above, "later" parts of the type specification can refer to O_1 and O_2 for asserting other properties; e.g. we can specify the struct-update effect of zero_y on the pointee by adding the constraint type $O_2.\mathrm{value} = O_1.\mathrm{value}\{.y = 0\}$ to the return type of zero_y. Often the C-type annotation for the Owned resource can be inferred from the pointer, in which case it can be omitted. Moreover, to optimise the common pattern of specifying properties of some pointer q's pointee, CN supports the notation $*q$ for doing so; CN supports the syntax "$\{*q\}$@start" and "$\{*q\}$@end" for referring to the pointee of q at the start or end of the function, instructing CN to "evaluate" $*q$ using the resource arguments ("pre-condition") or returned resources ("post-condition"), respectively. The $*q$ and '@' notation are surface-level features only, not present in the core type system. Shown below is the example in two equivalent versions using CN's concrete syntax. In CN, specifying Owned implicitly also asserts a "good" constraint for the pointee.

```
void zero_y (struct s* p)
/*@ requires let O1 = Owned<struct s>(p) @*/
/*@ ensures let O2 = Owned<struct s>(p) @*/
/*@ ensures O2.value == ({O1.value}@start){.y=0}@*/
  { p->y = 0; }
```

```
void zero_y (struct s* p)
/*@ requires Owned(p) @*/
/*@ ensures Owned(p) @*/
/*@ ensures *p == ({*p}@start){.y=0} @*/
  { p->y = 0; }
```

Fig. 1. CN specification for zero_y

2.7 Return and Function Types

We can now give the grammar of function types ft and return types rt. Here bt are *base types* (including mathematical integers integer, untyped pointers pointer, etc.), P are resource predicate names, including $\mathrm{Owned}\langle\tau\rangle$ for C-types τ, e and lc are expressions and constraint types, respectively, and x and O are variable names. Π and Σ bind computational variables in function and return types; CN has "effectful" let-bindings (with $-\!\!*$ and $*$) for asserting ownership of resources (second line) and iterated resources (third line), which we explain in §2.10, as well as "pure" let-bindings for terms; finally $\Rightarrow$ and $\wedge$ introduce constraint types into function and return types, respectively.

$$
\begin{aligned}
\mathit{ft} = {} & \Pi x : \mathit{bt}.\, \mathit{ft} \\
& \mid \text{let } O = P(e_1, \ldots, e_n) -\!\!* \mathit{ft} \\
& \mid \text{let } O = (\circledast_{i.\,G} P(e * i + k, e_2, \ldots e_n)) -\!\!* \mathit{ft} \\
& \mid \text{let } x = e;\ \mathit{ft} \\
& \mid \mathit{lc} \Rightarrow \mathit{ft} \\
& \mid \mathit{rt}
\end{aligned}
\qquad
\begin{aligned}
\mathit{rt} = {} & \Sigma x : \mathit{bt}.\, \mathit{rt} \\
& \mid \text{let } O = P(e_1, \ldots, e_n) * \mathit{rt} \\
& \mid \text{let } O = (\circledast_{i.\,G} P(e * i + k, e_2, \ldots e_n)) * \mathit{rt} \\
& \mid \text{let } x = e;\ \mathit{rt} \\
& \mid \mathit{lc} \wedge \mathit{rt} \\
& \mid I
\end{aligned}
$$

2.8 Resource and Logical Variable Inference

CN types include resources for reasoning about the safety of memory accesses and logical variables for abstracting over their values. For verifying C programs against these types, CN has to bridge a gap: CN uses first-class resources (splitting resource types from computational value types) while C programs are unaware of resource types; in checking C programs against CN types, the correct use of resources and the correct instantiation of logical variables has to be decided.

Requiring explicit annotations for both would require too much manual input, given the pervasive use of pointer accesses in C. On the other hand, any inference must be predictable. CN chooses a design where most resource inference is automatic, user instructions are required for packing/unpacking user-defined predicates (which can be inductive), and logical variables are inferred automatically. We illustrate CN's inference for `zero_y` from Fig. 1. (We slightly simplify the reasoning and skip over some steps in the Core program for presentation.)

(1) CN has typing contexts C, binding computational variables to base types; $\mathcal{L}$, binding logical variables to base types; $\mathcal{R}$, the set of available resources; and (as before) Φ, the set of constraints. For type checking a function, CN initially adds computational, logical, resource, and constraint arguments from the function type into the respective typing contexts; e.g. for let $O_1 = \mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p)$ in `zero_y`'s function type, CN adds O_1 to $\mathcal{L}$ and $\mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p)(O_1)$ to $\mathcal{R}$. CN then checks the function body.

(2) For any C function, the elaborated Core program first creates memory allocations for the local variables (including function arguments) and initialises their values. For `zero_y` it allocates and initialises one memory location, for the value of `p`. The implicit store for the initialisation has the return type

$$\Sigma u : \mathsf{unit}.\ \mathsf{let}\ O_s = \mathsf{Owned}\langle\mathsf{struct\ s\ ^*}\rangle(p\_l) * (O_s.\mathsf{value} = p) \wedge I$$

meaning it returns a computational value u of type unit, an Owned/points-to resource for the pointer p_l to the location of p in memory, with output argument O_s, and a constraint type specifying the pointee value $O_s.\mathsf{value}$ equals p. CN binds this return type into the context, adding, (for a fresh O_s) O_s to $\mathcal{L}$, $\mathsf{Owned}\langle\mathsf{struct\ s\ ^*}\rangle(p\_l)(O_s)$ to $\mathcal{R}$, and $(O_s.\mathsf{value} = p)$ to Φ.

(3) In Core, the L-value `p->y` of the assignment `p->y = 0` translates to a load of p (at location p_l) and a "member-shift" of this pointer. The typing rule for the load requires a resource $\mathsf{Owned}\langle\mathsf{struct\ s\ ^*}\rangle(e)(\_)$, for some pointer value e, for which $\bigwedge \Phi \Rightarrow (e = p\_l)$ is provable, and an arbitrary output argument. CN's resource inference scans $\mathcal{R}$ and finds $\mathsf{Owned}\langle\mathsf{struct\ s\ ^*}\rangle(p\_l)(O_s)$, from the allocation and initialisation step. The inferred resource's output argument determines the pointee value to be $O_s.\mathsf{value}$ (where $(O_s.\mathsf{value} = p) \in \Phi$), which the typing rule for the load returns: $\Sigma r : \mathsf{pointer}.\ (r = O_s.\mathsf{value}) \wedge I$.

(4) The typing rule for the `p->y` member-shift operation then returns a pointer p_{y} suitably offset from $O_s.\mathsf{value}$.

(5) The typing rule for the store of `0` to `p->y` requires a resource $\mathsf{Owned}\langle\mathsf{int}\rangle(e)(\_)$ for an e such that $\bigwedge \Phi \Rightarrow e = p_{\mathsf{y}}$ and arbitrary output argument. However, $\mathcal{R}$ instead contains a resource $\mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p)(\_)$, as per the function pre-condition. CN's resource inference includes automatically splitting or combining Owned resources based on their footprints and C-types: CN detects an inclusion of the footprint of the requested resource, $(p_{\mathsf{y}}, \mathsf{sizeof}(\mathsf{int}))$, and the available one $(p, \mathsf{sizeof}(\mathsf{struct\ s}))$ and splits the struct resource into resources for the members (and of padding bytes, where appropriate). Here this results in $\mathcal{R}$ containing (A) $\mathsf{Owned}\langle\mathsf{int}\rangle(p\_l\text{->}x)(\dots)$ and (B) $\mathsf{Owned}\langle\mathsf{int}\rangle(p\_l\text{->}y)(\dots)$, where $p\_l\text{->}x$ and $p\_l\text{->}y$ are CN expressions for the member-shifted pointer values. Now the resource inference finds resource (B), justifying the store. The typing rule for the store consumes it and returns a resource with updated output (pointee value).

(6) The Core function ends by de-allocating p_l. The typing rule (for "`kill`") requires a resource for a pointer e provably equal to p_l, inferred by CN as for the load at p_l earlier, and destroys it.

(7) Finally, CN checks against the function return type, with its let $O_2 = \mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p)$. CN now has to infer O_2 and the correct resource for this specification. CN's inference searches for a resource $\mathsf{Owned}\langle\mathsf{struct\ s}\rangle(e)(\_)$ for some e for which $e = p$ is provable and an arbitrary output. Due to the splitting of the struct-resource, $\mathcal{R}$ contains two resources (A and an updated B) for the members. The footprint analysis detects that the requested footprint is covered by these and combines them into a resource $\mathsf{Owned}\langle\mathsf{struct\ s}\rangle(p)(O_r)$ (for a padded struct, this step would also consume ownership of padding bytes); here O_r combines the values of A and B, and hence Φ appropriately relates O_r to the original resource output O_1 (i.e. O_r has member y updated to 0 compared to O_1). The found resource uniquely determines O_2 to be O_r. CN substitutes O_r for O_2

and proceeds with checking the remainder of the return type. Here, for instance, the constraint type (O_2.value = O_1.value$\{.y = 0\}$) under the substitution $[O_r/O_2]$ refers to O_r, for which Φ records the necessary information. (The resource reasoning in checking function calls with resource argument types is analogous to the "subtyping" check against the function return type just described.)

(8) At last, CN checks that the resource context is empty, with no unused resources.

Discussion. As illustrated above, CN's resource inference is not syntax-directed: e.g. a store to p requires ownership of some pointer that is provably-equal to p (rather than syntactically p). An earlier design envisaged syntax-directed resource typing, with all resources explicit in the C code; we decided against this to avoid a higher annotation burden for the user. Earlier versions of CN had syntax-directed unpacking of Owned resources: e.g. taking the L-value `p->y` in the example above as a hint to expand the struct resource. This correctly handled many examples but made verification brittle, e.g. pointer accesses could fail type checking due to irrelevant code changes earlier in the function. CN instead now relies on the footprint analysis described above. This makes resource inference computationally expensive — in the implementation, each resource request can involve several calls to Z3 — however, it allows CN to easily support pointer arithmetic. (As an optimisation we have implemented an expression simplifier that can obviate some SMT queries.)

2.9 Inductive Predicates

CN supports inductive resource predicates, a standard mechanism in separation logic for abstracting over the memory representation of data types [Reynolds 2002], but restricts the definition language based on a distinction between predicate inputs and outputs to guarantee reliable inference.

We illustrate them for a standard integer linked list data structure. (Note that CN does not currently support logical functions on lists; this example is for illustration only.) A linked list is a `struct node` pointer, where `NULL` encodes the empty list. A `struct node` comprises an element value `entry`, and a pointer `next` to the next `struct node` (or `NULL`) for the remainder of the list.

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

In separation logic, one might specify the predicate on the left below, linking a mathematical list l to the pointer p representing it:

$$
\begin{aligned}
&\mathsf{list}(p, l) = \\
&\quad (p = \mathsf{NULL} \wedge l = \mathsf{nil} \wedge \mathsf{emp}) \vee \\
&\quad (\exists n' : \mathsf{struct\ node}, l' : \mathsf{list}\langle\mathsf{integer}\rangle. \\
&\qquad l = \mathsf{cons}(n'.\mathsf{entry}, l') \wedge \\
&\qquad (p \mapsto_{\mathsf{struct\ s}} n' * \mathsf{list}(n'.\mathsf{next}, l')))
\end{aligned}
$$

```
predicate {list<integer> l} List (pointer p) {
  if ( p == NULL ) {
    return { l = nil<integer> }; }
  else {
    let Head = Owned<struct node>(p);
    let Tail = List(Head.value.next);
    return { l = cons(Head.value.entry, Tail.l) };
} }
```

This specifies that the list predicate holds if either: p is NULL, l is the empty list, and the heap is empty (emp); or there exists a node struct value n' and list value l' such that $l = \mathsf{cons}(n'.\mathsf{entry}, l')$, p points to n', and the list predicate holds for n'.next and the tail of the list l'.

The CN list predicate, shown on the right, is similar, but has a particular structure, enforced by CN, that guarantees reliable inference. The definition can be read roughly as follows: `List` has two arguments, an input `p` and an output `l`. The predicate "checks" whether `p` is NULL; if so, it returns/defines `l` as the empty list; otherwise it requires ownership of `p`, binding the outputs to `Head`, and (inductively) an instance of `List` for the tail of the list starting at the `Head` item's `next` pointer, binding the outputs to `Tail`. For the latter case it defines `l` as the cons of the `Head` item's value and the logical list value of the tail.

CN imposes the following restrictions on predicate definitions, to guarantee reliable inference. First, as described in §2.6, CN distinguishes between inputs and outputs of a resource, with the intuition that the inputs should uniquely determine the outputs. CN only allows specifications (function types or resource predicate definitions) to abstract over resource outputs, since for these the fact that they are uniquely determined guarantees successful inference. For the built-in $\mathrm{Owned}(p)(O)$ resource, CN regards pointer p as an input and pointee value O.value as an output: an owned pointer p uniquely determines the pointee value, since ownership is unique.

For predicates, CN requires the user to make this distinction. Determining arguments to be outputs will allow the user to abstract over these in other specifications, whereas input arguments have to be specified explicitly. The language of predicate definitions captures the intuition that fixing the inputs determines the outputs with a syntax *resembling function definitions*: inputs resemble C function arguments; outputs are specified using an anonymous-struct syntax in place of a C function's return type. Here, `p` is an input and `l` an output (with the intuition that fixing `p` uniquely determines the list value `l`). For CN to accept `l` as an output, it requires that `l` is indeed uniquely determined by the choice of inputs (here `p`); this is enforced by simply requiring that the definition does not depend on `l`: `l` is not in scope; it must be *defined* using return statements. Here the `NULL` case defines `l` to be `nil`, the non-`NULL` case as the cons of the head and the tail.

Second, CN does not allow general disjunction in predicate definitions; instead predicate definitions are ordered lists of guarded cases, where each case implicitly requires, in addition to its own guard, the negation of the preceding guards to hold. This ensures that the cases are mutually exclusive, and — combined with the above restriction that the cases do not depend on the output arguments — ensures that CN can determine, without back-tracking, which case applies based only on checking which guard applies for the given input arguments. The cases in this example are the same as in the separation logic predicate (`NULL` or not `NULL`), just encoded using if-then-else. Each case is a sequence of "specification statements", containing resource let-bindings (following the same syntactic restriction as return and function types described in §2.6), constraint type "assertions", and pure let-bindings for introducing names to expressions.

The resulting predicate definition has the same shape as the code a programmer might write for a recursive function or loop traversing the list: "check if the pointer is null or not; if null, done; if not, dereference the value of the head and recurse/loop with the `next` pointer."

While CN automates most resource inference, including the automatic packing/unpacking of resources $\mathrm{Owned}\langle\tau\rangle$ for complex C-types τ (structs and arrays), CN requires users to manually pack and unpack user-defined predicates, to avoid the need for back-tracking or unreliable heuristics in automating this. However, the restrictions CN places on predicate definitions, described above, mean that the only user input required for this is a CN "statement" of the form "pack $P(i_1, \ldots, i_n)$" or "unpack $P(i_1, \ldots, i_n)$", for a predicate name P and input arguments $i_1, \ldots, i_n$. The inference of the output arguments, determining which case of the predicate applies, and the inference required for that case's resource types are automatic.

2.10 Array Resources

Arrays are an essential feature of C that our type system needs to support. One could in principle encode arrays using inductive resource predicates, but this would not support computed accesses well: since such "random accesses" do not follow the inductive structure, lemmas or other user assistance would be required to handle them. Instead, CN supports arrays via iterated separating conjunction, with a scheme inspired by the work of Müller et al. [2016], but restricted to ensure quantifier-free SMT queries.

Consider, for instance, a pointer p with ownership for an int array of 100 items. CN can represent this as $\mathrm{Owned}\langle\mathrm{int}[100]\rangle(p)(V)$, i.e. a single Owned resource, of array type. Here the output V.value is an "SMT-array" map⟨integer, integer⟩, a function from integer indices to values. For reading or writing within an array, CN converts such array-typed resources, via an automatic unpack step, into an iterated separating conjunction, here:

$$\left(\mathop{\ast}_{i.\, 0 \le i < 100} \mathsf{Owned}\langle\mathsf{int}\rangle(p +_l (i * 4))\right)(V) \tag{A}$$

where $(+_l)$: pointer × integer → pointer and $p +_l n$ offsets p by n bytes (regardless of the pointee size). This asserts ownership for $p +_l (i * 4)$ for each array index i from 0 to 99, for the same output argument V. The $*$ iteration operator is treated as a quantifier that the inference mechanism handles automatically. In CN's implementation we use a concrete syntax resembling loops for specifying iterated resource types; e.g. the following specifies the same resource.

```
/*@ let V = each(integer i; 0 <= i && i < 100){Owned<int>(p + (i * 4))} @*/
```

CN constrains iterated resources to the shape $(\mathop{\ast}_{i.\,G} P(q +_l i * k, ia_2, \ldots ia_n)(O)$. Here i binds an integer-typed index; G is a boolean-typed expression, called the guard; and P a predicate name (i.e. Owned or a user-defined one) applied to some input arguments: a pointer argument $q +_l i * k$, for some expression q and constant k, that is a linear function of i (e.g. the linear progression $q +_l i * 4$ above) and possibly additional input argument expressions, which may depend on i. Finally, O is the record of outputs of the iterated resource, and so its fields have types lifted to maps: if predicate P has an output oarg of type bt, then O.oarg has type map⟨integer, bt⟩. Note that the guard does not need to be a simple inequality, so quantified resources need not form a contiguous array.

The constraints above are to simplify inference. To illustrate this, consider how a store `*q = 4`, via an `int` pointer `q`, might adjust (A). The typing rule for the store consumes an Owned⟨int⟩ resource from the context and returns one with the new value. In the presence of (A) in the context, there is an additional candidate "source" for the Owned⟨int⟩ resource. Since the resource pointer in (A) is linear, the inference can compute exactly the candidate index $i_q = (\mathsf{cast}_{\mathsf{integer}}(q) - \mathsf{cast}_{\mathsf{integer}}(p))/\mathsf{sizeof}\langle\mathsf{int}\rangle$ which might result in a matching resource [Müller et al. 2016]. CN considers it a match if the iterated resource's guard G (here $0 \le i < 100$) holds for this i_q (according to the SMT solver). If so, CN extracts ownership for i_q, by adjusting the iterated resource (A) to have the guard $0 \le i < 100 \wedge i \ne i_q$ and splitting out the single instance at i_q as a separate resource. The extracted Owned resources for i_q corresponding to q allow the inference to succeed.

Note that this works whenever q is provably a pointer to a cell of the array. Obviously this works when q is constructed as an array offset syntactically, but CN also supports cases where pointers into an array are read from other structures in memory or computed by pointer arithmetic, and our buddy allocator example in §3 requires this.

In addition to splitting elements from arrays, CN can infer the reverse step, joining a single element to an array. For instance, this may be needed for passing the modified array to another function following the store `*q = 4`. To infer a single resource for the whole array, with a required guard G_{req} of $0 \le i < 100$, CN first examines the non-quantified resources of matching type in the context (here those of Owned⟨int⟩ type). For each such resource R with pointer r, CN computes the index i_r at which r would appear in the array, and checks whether G_{req} holds for i_r (using the SMT solver); when it does, i_r corresponds to a required index within the array and CN "collects" it: removing R from the context and updating G_{req} to $G_{req} \wedge i \ne i_r$ to record that i_r is no longer needed. After all non-quantified resources have been scanned, a single quantified resource must be found in the context, with matching address and a guard covering the remaining G_{req}. This quantified resource is combined with the already-collected resources, into a single resource whose output argument can be described via array updates. For instance, in our simple example, the combined resource would be $\mathsf{Owned}\langle\mathsf{int}[100]\rangle(p)(V')$ with $V'.\mathrm{value} = V.\mathrm{value}[i_q := 4]\}$.

Discussion. CN's approach to iterated separating conjunction is based on the work of Müller et al. [2016]. However, we do not allow combining the ownership of multiple iterated resources, to ensure the outputs of the combined resource can be described in a quantifier-free way. To our understanding, the scheme of Müller et al. is more general, permitting such merging, however, in a system that requires quantified SMT queries (handled via triggers), which we aim to avoid. Moreover, their (elegant) resource inference loop is optimised to only ask the SMT solver once, and otherwise "symbolically subtracts" required ownership from existing resources, simplifying the resulting terms where possible. We experimented with this style, but found that it lead to syntax explosions, especially in the guard component of iterated resources. With our approach, we obtained more predictable performance, albeit at the cost of more SMT queries.

2.11 Lemmas

As discussed in §2.4, CN reduces refinement typing to decidable propositional logic reasoning, while still supporting code and specifications with non-linear arithmetic, and quantified constraint types. It does so by mapping non-linear arithmetic to uninterpreted functions, and using a default quantifier instantiation strategy and manual quantifier instantiation by the user. Where reasoning requires knowledge about uninterpreted functions, which are opaque to CN and Z3, or CN's quantifier instantiation methods are insufficient, CN requires user assistance: users need to manually "fill the gaps" in the proof by capturing the relevant reasoning steps in *lemmas*.

```
void lemma_power2_def(int y)
/*@ trusted @*/
/*@ requires y >= 0 @*/
/*@ ensures (power_uf(2,y+1)) == (2*(power_uf(2,y))) @*/
{}
```

Lemmas in CN are just C functions with empty body and a CN type annotation that states the relevant property. For instance, as part of a verification the user may require CN to know the fact $2^{y+1} = 2 * 2^{y}$, for an uninterpreted power_uf function and for a concrete instance of y from the code. This can be captured with the lemma shown above. The user must explicitly "invoke" this lemma via a function call that specifies the value y. The `trusted` annotation tells CN that this type signature should not be proven in CN but instead exported to a theorem prover. CN generates a Coq theory file which defines the proposition to be proven for each lemma in the C source. These are gathered in a module specification which also takes the uninterpreted functions as parameters. If the user can instantiate this module, i.e. provide interpretations of the functions and proofs of the propositions, then the verification is complete. Currently we support "pure" lemmas, with implications involving only constraint types. These can be encoded in Coq using only integers, functions and tuples. In the future we plan to add support for lemmas about resources, so resource equivalences that would be difficult to infer automatically can be proved manually.

It is an open empirical question of how often in practical systems C verification one would need lemmas, for reasoning about non-linear arithmetic and quantifiers, and whether this mechanism is sufficiently convenient, but at least in the buddy allocator case study of §3, it suffices.

2.12 Logical Variables

As discussed in §2.6 and §2.9, CN restricts the use of logical variables in type specifications to guarantee that they can be inferred reliably. Assume for the moment that CN allowed arbitrary logical variable bindings via ∀ and ∃ in function and return types, respectively, and consider the following type ft_1 for a function f:

$$\begin{aligned} ft_1 = \Pi p : \mathsf{pointer}.\ & \forall q_1 : \mathsf{pointer}. \forall v_1 : \mathsf{integer}. (\mathsf{Owned}\langle\mathsf{int}\rangle(q_1)\{.\mathsf{value} = v_1\}) \multimap \\ & \forall q_2 : \mathsf{pointer}. \forall v_2 : \mathsf{integer}. (\mathsf{Owned}\langle\mathsf{int}\rangle(q_2)\{.\mathsf{value} = v_2\}) \multimap \\ & (lc_1[q_1, v_1]) \Rightarrow (lc_2[q_2, v_2]) \Rightarrow rt \end{aligned}$$

whereby f takes a computational argument p of pointer type, and two Owned resources, for arbitrary pointers q_1, q_2 and pointee values v_1, v_2, satisfying the constraint types lc_1 and lc_2. (Perhaps the latter relate q_1 and q_2 to known concrete values, such as p — how else could the Owned resources for those pointers be useful to f?) This specification would be problematic for inference. Consider a function call $f(r)$; after checking r against the computational argument type, CN would have to infer the instantiation of q_1 and v_1 and the first Owned resource. If the resource context at this point contained two Owned⟨int⟩ resources, CN would not be able to decide which one to pick, and how to instantiate q_1 and v_1, *without backtracking*: CN would need to try a choice, then infer the second resource, instantiate q_2 and v_2 accordingly, and check lc_1 and lc_2 before "knowing" whether the choice was correct, potentially having to re-wind and try the second choice.

On the other hand, given the type ft_2 for f, shown below, where q_1 and q_2 are replaced by known pointer values $p +_l 4$ and $p +_l 8$ (p offset by 4, resp. 8, bytes using the notation from §2.10), the instantiation of the remaining logical variables, v_1 and v_2, is easy: first, find the unique Owned⟨int⟩ resource with a pointer provably equal to $p +_l 4$ (if any exists) and instantiate v_1 to match its value; second, do the same for $p +_l 8$ and instantiate v_2.

$$\begin{aligned} ft_2 = \Pi p : \mathsf{pointer}.\, &\forall v_1 : \mathsf{integer}.\, (\mathsf{Owned}\langle\mathsf{int}\rangle(p +_l 4)\{.\mathsf{value} = v_1\}) \multimap \\ &\forall v_2 : \mathsf{integer}.\, (\mathsf{Owned}\langle\mathsf{int}\rangle(p +_l 8)\{.\mathsf{value} = v_2\}) \multimap \\ &(lc_1[p +_l 4, v_1]) \Rightarrow (lc_2[p +_l 8, v_2]) \Rightarrow rt \end{aligned}$$

For predictable type checking it is desirable to avoid the backtracking needed for ft_1 and guarantee that logical variables are immediately correctly resolved, as in ft_2. What makes the use of logical variables q_1 and q_2 in ft_1 problematic and v_1 and v_2 in ft_2 not? The logical/"non-computational" part of ft_1 amounts to an *imprecise separation logic predicate* [Reynolds 2008]: e.g.

$$\forall q_1 : \mathsf{pointer}.\, \forall v_1 : \mathsf{integer}.\, (\mathsf{Owned}\langle\mathsf{int}\rangle(q_1)\{.\mathsf{value} = v_1\})$$

can be satisfied by an arbitrary owned heap cell, because q_1 is unknown (similarly for q_2). This means any Owned⟨int⟩ resource in the context can satisfy this type, and q_1 and v_1 are underspecified. In contrast, in ft_2 the concrete pointer $p +_l 4$ of the first resource type uniquely identifies a heap cell; hence, only one (if any) resource from the context $\mathcal{R}$ can satisfy this type. Since ownership of Owned resources is unique, this uniquely determines the instantiation of the pointee value v_1. (The second resource and v_2 are analogous.) More generally, each argument of a resource can be assigned a mode, *input* or *output*, such that fixing the inputs of a resource uniquely determines the outputs; in the case of points-to/Owned the pointer is an input and the pointee an output.

Logical variables should be restricted so they can always be inferred based on resource types, as illustrated in the case of ft_2: each resource type in the specification should determine a resource in the typing context (if any) that satisfies it and thereby uniquely determine the instantiation of any logical variables in the resource type. Note that a typing context may contain multiple interchangeable resources (i.e. two instances of the same predicate applied to provably-equal arguments), which is possible, for instance, with predicates that do not assert ownership of memory; this is fine, since the choice between them leads to provably-equal instantiations of logical variables.

In a setting without CN's resource-let-bindings from §2.6 and §2.9 but with general ∀ and ∃ logical-variable binders, one could achieve such a restriction of logical variables by defining a *mode check* function for types, MC, which ensures:

- that each logical variable will be resolved as an output of some resource,
- that each constraint type only depends on logical variables that are resolved "at that point",
- that each resource's inputs are resolved "at that point".

Below is a possible definition of a mode check for function types: MC takes sets v_r and v_u of resolved and unresolved variables, both initially empty, and a function type ft, and checks the use of unresolved variables, by induction on the structure of ft. Here FV e is the set of free variables in e. (1) explicitly passed run-time values are considered resolved; (2) logical variables are initially unresolved; (3) refinement types must not refer to unresolved variables; (4, 6) neither should resource inputs; (5) resource outputs resolve logical variables; (7) once the return type rt is "reached", all logical variables should be resolved. In this definition, v_r is passed for illustration purposes only, the code does not actually use it. Return types and predicate definitions can be checked analogously.

$$
\begin{array}{ll}
1 & \mathsf{MC}\ v_r\ v_u\ (\Pi x : bt.\,ft) = \mathsf{MC}\ (v_r \cup x)\ v_u\ ft \\
2 & \mathsf{MC}\ v_r\ v_u\ (\forall x : bt.\,ft) = \mathsf{MC}\ v_r\ (v_u \cup x)\ ft \\
3 & \mathsf{MC}\ v_r\ v_u\ (lc \Rightarrow ft) = \mathsf{FV}\ lc \cap v_u = \emptyset \wedge \mathsf{MC}\ v_r\ v_u\ ft \\
4 & \mathsf{MC}\ v_r\ v_u\ (re \twoheadrightarrow\!\!\ast\ ft) = \\
5 & \qquad \mathsf{let}\ v_r' = \mathsf{outputs}\ re \cap v_u \mathsf{in} \\
6 & \qquad \mathsf{FV}\ (\mathsf{inputs}\ re) \cap v_u = \emptyset \wedge \mathsf{MC}\ (v_r \cup v_r')\ (v_u \setminus v_r')\ ft \\
7 & \mathsf{MC}\ v_r\ v_u\ rt = v_u = \emptyset
\end{array}
$$

Inspecting the shape of the types ft that pass this check, one can find that those all have the property that every logical variable x introduced via $\forall$, is first "mentioned" as an output of a resource (since otherwise the empty-conjunction checks in (3) and (6) fail). This observation allows CN to enforce mode correctness in the use of logical variables in a simpler way: instead of general $\forall$ and $\exists$ binders for logical variables, CN instead uses the resource-let-binding syntax described in §§2.6,2.9. Resource-let-bindings let $O = P(i_1, \ldots, i_n)$, for some predicate P and input $i_1, \ldots, i_n$, simultaneously:

- introduce a logical variable O, for the record of outputs $\{.\mathsf{f}_1 = o_1, \ldots, .\mathsf{f}_m = o_m\}$ of P, and
- assert ownership of P applied to inputs $i_1, \ldots, i_n$ and outputs $o_1, \ldots, o_m$.

This *syntactically* links the introduction of logical variables to their use as resource outputs and captures the types accepted by MC. It does so by enforcing *mode-correctness via variable scoping*: MC rejects types that introduce a logical variable via $\forall$ or $\exists$ and use it in a constraint type (3) or resource input (6) before it is resolved as a resource output. With CN's resource-let-binding syntax this is just a scoping violation: a logical variable O only enters the scope when used as an output argument. Hence, mode failures can be explained to users in term of familiar variable scoping.

Note that our restriction does not preclude the use of logical variables in resource inputs; e.g.

$$
\begin{aligned}
\Pi p : \mathsf{pointer}.\ \mathsf{let}\ O_1 &= \mathsf{Owned}\langle\mathsf{int}^*\rangle(p) \twoheadrightarrow\!\!\ast \\
\mathsf{let}\ O_2 &= \mathsf{Owned}\langle\mathsf{int}\rangle(O_1.\mathsf{value}) \twoheadrightarrow\!\!\ast \ldots
\end{aligned}
$$

specifies the type of a function that takes an `int`-pointer-pointer p, ownership for p, and ownership for p's pointee O_1.value; the latter is asserted using the first resource's output as an input for the second. When checking a function call against this type, O_1 is resolved after inferring the first resource, at which point the second resource's inputs are fixed, so that its inference can proceed.

3 BUDDY ALLOCATOR

We now demonstrate CN's usability in the verification of pKVM's buddy allocator, used after initialisation for managing the memory it uses for various page tables. It was written by a conventional development team at Google as part of Android/Linux. The buddy allocator code, together with the required dependencies has 364 non-comment lines of C code (leading to 6169 lines of Core). In our formalisation, we add 417 lines of function and loop specifications and 78 lines of in-function instrumentation, plus 249 lines for auxiliary definitions and 165 for lemma statements and 1219 lines of Coq for their proofs, totalling 5.85× the code size including Coq proofs and 2.50× excluding Coq proofs. Checking the buddy allocator in CN takes 141s on a standard laptop (2GHz Quad-Core Intel Core i5, 16GB ram). We verify a version of the buddy allocator from October 2020[1]. The verification involves difficult ownership reasoning, pointer arithmetic into an aliasing

[1]**original:** https://android-kvm.googlesource.com/linux/+/39111fc40453747f8213cf9ef4337448d3c6197d/arch/arm64/kvm/hyp/nvhe/page_alloc.c **formalisation:** in the online materials;

data structure (including XOR of pointer representation bits), iterated resources, and complex universally quantified constraints.

CN does not currently support concurrency; we comment out locks and the buddy allocator's lock type (which also requires unions) — a standard separation-logic treatment of locks would handle these — and Linux `READ_ONCE` and `WRITE_ONCE` operations. We also make the following light changes: (1) We make minor additions to the initial meta-data setup (conveniently setting initial refcounts and page orders as assumed by auxiliary functions), replacing a memset zero'ing out this data (for which CN would have required (easily added) support for converting Owned resources of a group of representation bytes to an Owned resource of the represented C-type). (2) The main allocator invariant is associated with a "pool" pointer, pointing to certain allocator meta-data. For proof convenience we add this pool pointer as an extra argument to some functions. (3) We initialise some variables at declaration time to overcome a limitation of CN's annotation language. (4) We simplify the macro definition for the `min` minimum operation. (5) Finally, in the allocator initialisation, we skip the verification of one auxiliary arithmetic function `get_order`, with non-linear integer arithmetic, whose value the verification does not depend on; we merely "trust" it to return values greater than 0.

3.1 Overview of the Allocator Implementation

The allocator follows a standard buddy scheme. It allocates blocks, each of size 2^o 4k pages of some *order* $o \in 0..$`MAX_ORDER`$=11$, and with its physical address 2^o page-aligned. The allocatable memory is partitioned into pools, each a contiguous range of pages, with metadata in a `struct hyp_pool` comprising a lock (omitted for this verification), its range of physical start and end address, its maximum order, and, for each order, the head of a list of its free blocks at that order. This is a standard Linux-kernel circular doubly-linked list, empty iff the head points to itself. The *buddy* of a block B of order o is the (unique) adjacent order-o block that can be merged with B into a block of order $o + 1$ with 2^{o+1} page-aligned physical address (if the buddy is within the range of the pool).

```
struct hyp_pool {
  /* hyp_spinlock_t lock; */
  struct list_head free_area[MAX_ORDER];
  phys_addr_t range_start;
  phys_addr_t range_end;
  unsigned int max_order;
};
struct list_head {
  struct list_head *next, *prev;
};
```

The buddy physical address is calculated by flipping bit $o + 12$ of B's address. If a block of order o is requested from a pool and the corresponding free list does not contain one, the allocator splits a higher-order block as needed; conversely, when a block is returned, it coalesces it with any free buddy blocks of the same or higher order.

The allocator's main state is the `vmemmap`, an array of per-page `struct hyp_page` metadata. This includes a refcount; the order $o \in 0..$`MAX_ORDER`$- 1$ if this is the initial page of any free or allocated block, or `HYP_NO_ORDER` otherwise; a pointer to the associated pool; and a `list_head` node. The free lists are maintained as intrusive lists, with the list pointers embedded within the linked structures, a standard idiom in Linux. The buddy allocator also accesses a global signed integer `hyp_physvirt_offset` to map between hypervisor physical and virtual addresses.

```
struct hyp_page {
  unsigned int refcount;
  unsigned int order;
  struct hyp_pool *pool;
  struct list_head node;
};
```

The API functions are `hyp_pool_init`, to initialise the allocator, `hyp_alloc_pages`, to request a block of a particular order, `hyp_get_page`, to increase the refcount of an allocated block, and `hyp_put_page`, to decrement a block's refcount and, if zero, return block ownership. (As noted earlier, compared to the original, we add the pool pointer to the get and put functions.)

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

```
void *hyp_alloc_pages(struct hyp_pool *pool, unsigned int order);
void hyp_get_page(struct hyp_pool *pool, void *addr);
void hyp_put_page(struct hyp_pool *pool, void *addr);
```

3.2 Verification Approach

The Specification. The API functions have types phrased in terms of Hyp_pool, the allocator's main invariant, defined shortly. We give the following specifications (omitting some details). Function `hyp_pool_init` takes a start-of-the-range page index and a non-zero number of pages; it requires some well-formedness conditions to hold (such as range, alignment and pointer disjointness constraints), and requires ownership of the pool meta-data, the pool's range of the `vmemmap`, and of (not necessarily zeroed) order-0 blocks for the pool's range. It then returns the Hyp_pool invariant for this range. Function `hyp_alloc_pages` takes a pool pointer and an order, and requires Hyp_pool; it returns the Hyp_pool predicate and either a pointer to the virtual address of a freshly allocated block at `order` together with appropriate block ownership, or a NULL pointer without ownership. Function `hyp_get_page` takes a pool pointer and a block virtual address whose corresponding physical address is in the pool's range, requires Hyp_pool and that the block's initial page has `refcount` non-zero and less than $2^{31} - 1$ (so it can be incremented); it returns the Hyp_pool predicate. Finally, `hyp_put_page` takes a pool pointer and a virtual block address `addr` in its range, requires Hyp_pool, that the physical address corresponding to `addr` is page-aligned and non-NULL, and that the block's initial page has a non-zero `refcount`; it additionally requires ownership of a block of memory for `addr`, at the order of the block's initial page, if the current `refcount` is 1 (and so will be decremented to 0); it then returns Hyp_pool.

The main difficulty in the verification is handling pKVM's use of computed accesses and pointer aliasing into the `vmemmap`. Its entries are accessed using two kinds of pointers:

(1) `prev` or `next` linked-list pointers into the `vmemmap`;
(2) for a (suitably aligned) physical page address p, computed access to its meta-data at `vmemmap` index $p/(2^{12})$.

Both kinds of pointers are dereferenced by pKVM, the safety of which has to be verified. However, CN cannot give the (unique) ownership to both kinds of pointers simultaneously. Instead, we choose a strategy based on iterated resources. We assert ownership once, of all `vmemmap` entries for a pool's range of physical memory, and have additional constraint types that guarantee that each `struct hyp_page` accessed by pKVM falls in this range. The main invariant, the resource predicate Hyp_pool, captures these and various other conditions. It takes as arguments the pool pointer `pool_l`, the vmemmap pointer `vmemmap_l` and the current value `physvirt_offset` of `hyp_physvirt_offset` and is defined as follows (omitting predicate outputs and some details):

```
predicate ... Hyp_pool (pointer pool_l, pointer vmemmap_l, integer physvirt_offset) {
  let P = Owned<struct hyp_pool>(pool_l);                                        /*A*/
  let start_i = P.value.range_start / 4096;
  let end_i = P.value.range_end / 4096;
  let off_i = physvirt_offset / 4096;
  let V = each(integer i; (start_i <= i) && (i < end_i))                         /*B*/
            {Owned<struct hyp_page>(vmemmap_l + i*32)};
  assert (each(integer i; (start_i <= i) && (i < end_i))                         /*C*/
            {vmemmap_b_wf (i, vmemmap_l, V.value, pool_l, P.value)});
  let R = each(integer i; (start_i <= i + off_i) && (i + off_i < end_i)          /*D*/
              && (((V.value)[i+off_i]).refcount == 0)
              && (((V.value)[i+off_i]).order != (hyp_no_order ())))
            {ZeroPage(((pointer) 0) + i*4096, 1, ((V.value)[i+off_i]).order)};
  ...
}
```

(A) Resource: `pool_l` is an owned `struct hyp_pool` pointer; the output is bound to the name `P` (and so `P.value` is the logical struct value).

(B) Resource: the allocator pool owns the iterated separating conjunction of all `vmemmap` indices i from `start_i = P.value.range_start/4096` to `end_i = P.value.range_end/4096`, where $4096 = 2^{12}$ = `PAGE_SIZE` and 32 = `sizeof<struct hyp_page>`. This binds `V`, so `V.value` is the logical `vmemmap` meta-data array.

(C) Constraint: for any `vmemmap` index `i` from `start_i` to `end_i` some properties hold, including the following two that are specified as part of the user-defined function `vmemmap_b_wf`:
- Its linked-list pointers, if non-empty, point to `vmemmap` entries in the same range or into the pool's `free_area` array. This condition guarantees that we have the ownership required for dereferencing these, either from (A) or (B). Note that this means that we do not track the contents of the linked list; we merely assert some constraints each linked list `prev`/`next` pointer has.
- Its physical address, $p = i * 2^{12}$, is 2^{12+o}-aligned, for the order o=`(V.value[i]).order` (specified with uninterpreted functions due to the non-linear arithmetic). The condition ensures that the computed address of the buddy of p is in the pool's range, meaning that (B) includes the ownership required for accessing the buddy's meta-data.

(D) Resource: the allocator owns all free pages, as an iterated resource holding ownership of `ZeroPage` for each page address $i * 4096$ wherever the page address is in the right range and its `vmemmap` entry, in `V.value`, has 0 refcount and a non-`HYP_NO_ORDER` order. The resource `ZeroPage` captures ownership of a zeroed block of memory, of the page's order. It is phrased in terms of virtual addresses (so these can be accessed via C pointers). To this end, i is the index for the virtual address and we use `i + off_i` to translate this to the corresponding physical address index, which we can use to index the `vmemmap`, `V.value`.

Note that simpler strategies, such as using an inductive list predicate for the ownership of `vmemmap` entries, do not support the computed `vmemmap` access that the code relies on.

3.3 Example Function

We now walk through an example function to illustrate the reasoning required in the verification. The function `__hyp_attach_page` returns ownership of a block B to the allocator, coalescing it with adjacent free blocks. We show the un-annotated source code below.

```
1  static void __hyp_attach_page(struct hyp_pool *pool,
2                                struct hyp_page *p) {
3    unsigned int order = p->order;
4    struct hyp_page *buddy = NULL;
5    memset(hyp_page_to_virt(p),0,PAGE_SIZE << p->order);
6    p->order = HYP_NO_ORDER;
7    for (; (order + 1) < pool->max_order; order++) {
8      buddy = __find_buddy_avail(pool, p, order);
9      if (!buddy)
10       break;
11     list_del_init(&buddy->node);
12     buddy->order = HYP_NO_ORDER;
13     p = min(p, buddy);
14   }
15   p->order = order;
16   list_add_tail(&p->node, &pool->free_area[order]); }
```

The function arguments are a pool pointer and a pointer `p`, to the `vmemmap` meta-data for the initial page in B, within this pool. Its body sets `order` to that of `p` (3), zeroes B (5), sets `p`'s order to `HYP_NO_ORDER` (6), and loops. In each iteration, `__hyp_attach_page` checks whether `p`'s buddy at `order` starts a free block (8,9), and, if so, pulls out the buddy's block, by deleting it from the free list (11) and setting its order to `HYP_NO_ORDER` (12). It then coalesces the blocks of `p` and the buddy, by making `p` the initial page of the combined block, their minimum (13); finally, it increments `order` (7) to reflect this coalescing, and loops. It loops until the order is maximal or no free buddy is

```
1  for (; (order + 1) < pool->max_order; order++)
2  /*@ inv let p_i2 = (((integer) p) - __hyp_vmemmap) / 32 @*/
3  /*@ inv let Z = ZeroPage((pointer) ((p_i2 * 4096) - hyp_physvirt_offset), 1, order) @*/
4  /*@ inv let OP = Owned(pool) @*/
5  /*@ inv let hyp_vmemmap = (pointer) __hyp_vmemmap @*/
6  /*@ inv let start_i2 = (*pool).range_start / 4096 @*/
7  /*@ inv let end_i2 = (*pool).range_end / 4096 @*/
8  /*@ inv let off_i = hyp_physvirt_offset / 4096 @*/
9  /*@ inv let V2 = each (integer i; start_i2 <= i && i < end_i2)
10                       {Owned<struct hyp_page>(hyp_vmemmap+(i*32)) } @*/
11 /*@ inv let p_page = V2.value[p_i2] @*/
12 /*@ inv let p_page_tweaked2 = (p_page){.order = order} @*/
13 /*@ inv each(integer i; start_i2 <= i && i < end_i2)
14            {vmemmap_b_wf(i, hyp_vmemmap, V2.value[p_i2 = p_page_tweaked2], pool, *pool)} @*/
15 /*@ inv each(integer i; 0 <= i && i < ((*pool).max_order))
16            {freeArea_cell_wf(i, hyp_vmemmap, V2.value, pool, (*pool))} @*/
17 /*@ inv hyp_pool_wf(pool, *pool, hyp_vmemmap, hyp_physvirt_offset) @*/
18 /*@ inv let R = each(integer i; start_i2 <= (i + off_i) && (i + off_i) < end_i2
19                       && (V2.value[i + off_i]).refcount == 0
20                       && (V2.value[i + off_i]).order != (hyp_no_order ()))
21                      { ZeroPage(((pointer) 0) + (i*4096), 1, (V2.value[i+off_i]).order) } @*/
22 /*@ inv 0 <= order; order+1 <= (*pool).max_order @*/
23 /*@ inv cellPointer(hyp_vmemmap,32,start_i2,end_i2,p) @*/
24 /*@ inv (p_page.refcount) == 0; (p_page.order) == (hyp_no_order ()); (p_page.pool) == pool @*/
25 /*@ inv (p_page.node.next) == &(p->node); (p_page.node.prev) == &(p->node) @*/
26 /*@ inv order_aligned(p_i2,order) @*/
27 /*@ inv (p_i2 * 4096) + (page_size_of_order(order)) <= (*pool).range_end @*/
28 /*@ inv each(integer i; {p_i}@start < i && i < end_i2)
29            {((({V.value[i]}@start).refcount == 0) || ((V2.value[i]) == {V.value[i]}@start)} @*/
30 /*@ inv {__hyp_vmemmap} unchanged; {hyp_physvirt_offset} unchanged; {pool} unchanged @*/
31 /*@ inv ({*pool}@start){.free_area = (*pool).free_area} == *pool @*/
32 {
33     buddy = __find_buddy_avail(pool, p, order);
34     if (!buddy)
35             break;
36     /*CN*/instantiate vmemmap_b_wf, hyp_page_to_pfn(buddy);
37     /*CN*/unpack ZeroPage (hyp_page_to_virt(p), 1, order);
38     /*CN*/unpack ZeroPage (hyp_page_to_virt(buddy), 1, order);
39     /*CN*/lemma_attach_inc_loop(*pool, p, order);
40     /*CN*/lemma2(hyp_page_to_pfn(p), order);
41     /*CN*/lemma_page_size_of_order_inc(order);
42     /*CN*/if ((buddy->node).next != &pool->free_area[order])
43     /*CN*/  instantiate vmemmap_b_wf,
44                   hyp_page_to_pfn(container_of((buddy->node).next, struct hyp_page, node));
45     /*CN*/if ((buddy->node).prev != &pool->free_area[order])
46     /*CN*/  instantiate vmemmap_b_wf,
47                   hyp_page_to_pfn(container_of((buddy->node).prev, struct hyp_page, node));
48     /*CN*/if ((buddy->node).prev != (buddy->node).next);
49     list_del_init(&buddy->node);
50     buddy->order = HYP_NO_ORDER;
51     p = min(p, buddy);
52     /*CN*/pack ZeroPage (hyp_page_to_virt(p), 1, order + 1);
53 }
```

Fig. 2. Annotated loop body from __hyp_attach_page. In-function annotations are prefixed with /*CN*/. Along with __hyp_extract_page this is the most difficult function, with a high annotation overhead.

found. After the loop, `__hyp_attach_page` adds the coalesced block into the allocator by setting p's order to `order` (15) and adding p to the free list (16).

We now describe the verification steps, just for "non-breaking" execution of the loop. Fig. 2 shows the fully CN-annotated loop body (in the following, all line numbers refer to this figure). The invariant asserts (roughly):

(I1) Ownership of ZeroPage, a zeroed block at the page address corresponding to p, of order order: see line 3. Initially this is the ownership of B; each iteration "grows" the ownership, by coalescing adjacent free blocks.

(I2) The Hyp_pool invariant holds — at least almost: since p is "being processed" in the loop, some of its meta-data does not satisfy certain conditions of `Hyp_pool` until `__hyp_attach_page` is fully executed. Hence lines 4–21, the details of `Hyp_pool` are expanded and adjusted where `p` needs to be treated specially. (The remaining lines 22–31 record some additional facts about `p`, or state that certain parts of the state are unchanged by the loop body; we skip the discussion of these details in the following description.)

The verification of the loop body consists of three main proof goals: proving safety, and proving that invariants I1 and I2 are re-established.

Safety. CN has to verify the safety of the access to the buddy pointer (49,50). Invariant I2 asserts ownership of this pool's `vmemmap` range; as per `__find_buddy_avail`'s post-condition, `buddy` is in this range. Hence, CN's inference detects that it can break the Owned resource for `buddy` out of the `vmemmap` iterated separating conjunction, justifying (50). The list deletion (49) is more difficult since it requires ownership of not just the buddy but also its `prev` and `next` pointers (recall that we do not use a linked list resource predicate). We now manually `instantiate` `vmemmap_b_wf` of I2 for buddy, to conclude that `prev` and `next` are either `vmemmap` pointers or pointers to the pool's `free_area` (line 36; `hyp_page_to_pfn` returns the buddy's `vmemmap` index, at which we instantiate `vmemmap_b_wf`). Either way we have ownership. Since it can be from different "sources", however, we have to manually distinguish these cases (which we do using C's if-then-else; lines 42, 45). Additional difficulty arises from the fact that the free lists are *circular*. That means `prev` and `next` could alias, and the specification of `list_del_init` treats this case specially, since it then only receives ownership of one resource; and we have to make another case distinction at the call site (48). With the guidance to distinguish these cases, CN's resource inference automatically infers the resources for the list access, breaking one or two Owned resources out of the `vmemmap` iterated resource or the pool's `free_area` resource, as needed.

I1. By assumption we have ownership of the block for the page address corresponding to `p`, at `order`; we have to prove ownership of a block for the updated `p` and incremented `order`. Combining the post-condition of `__find_buddy_avail` and the `vmemmap_b_wf` condition for `buddy`, instantiated earlier, we know that `buddy` has `refcount` 0 and `order ≠ HYP_NO_ORDER`. According to (I2) we then have ownership of `ZeroPage` for `buddy`'s page address at order `order`, which we should combine with the block ownership by assumption (I1). We manually `unpack` both (37,38) and re-pack a single combined block (52) to establish the loop invariant (I1). The ability to combine these resources requires showing the adjacency of the page addresses of `buddy` and `p` for blocks at `order` — non-linear integer arithmetic reasoning, which we state in lemmas and invoke here to finish the proof (40, 41). The lemmas are mostly simple arithmetic facts about exponentiation and alignment, which we export and manually prove in Coq.

I2. Finally, we have to show invariant (I2) holds. This mainly requires proof that constraint types such as `vmemmap_b_wf` are correctly re-established. This proof is a combination of automatic refinement reasoning, exercising CN's default quantifier instantiation to prove that the universally quantified constraint types in the assumption imply the universally quantified constraint types in the proof goal, and manual instantiation at additional relevant indices — here instantiating `vmemmap_b_wf` for `buddy` (36, as before) and the buddy's `prev` and `next` pointers (46,43), as well as lemmas to close gaps in the proof of invariants involving non-linear integer arithmetic (39).

4 FORMALISATION

In this section, we formalise and prove type safety for "kernel CN", which is essentially ordinary CN with no type and resource inference. In particular, we assume that all universal quantifiers are explicitly instantiated, that all existential quantifiers have explicit witnesses, and all resource manipulations have proof terms with linear/substructural types. However, we do not require proof terms for the logical properties, since by construction all of the entailments fall into the SMT fragment. Since our inference algorithm can be extended to an elaboration algorithm producing a fully-annotated program (which we have so far formalised for iterated resource manipulation, though not footprint analysis), kernel CN could serve as an intermediate representation for the CN compiler. Moreover, the lack of inference makes it a simpler language to prove type safety for.

Kernel CN is a calculus in A-normal form, with a bidirectional type system. Since we handle most of C (via Core), the entire system is large. We only present the highlights here; the full details, and a discussion of minor differences compared to the implementation, are in the online materials.

4.1 Types and Terms

As mentioned earlier, CN programs have both computational and logical terms. Every such term, computational or ghost, has a *base type* β, which are things like unit, booleans, (mathematical) integers, locations, and records of other base types. Each C type τ is mapped to a corresponding base type – for example, `to_base(int*) = pointer`. Logical terms are variously referred to as *term*, *ptr* (for pointers), *value* (for pointees), *iarg* (for input-arguments), *oarg* (output-arguments, of type record or array of records), and *iguard* (for boolean guards of iterated resources).

In Figure 3, we give the grammar of resource types (i.e., separation logic predicates) and resource terms (the proof terms used by the kernel Core typechecker). The standard resources *res* can be an empty heap `emp`, a boolean condition *term*, the separating conjunction $res_1 * res_2$, an existential type $\exists\, y{:}\beta.\ res$, and the disjunction `if` *term* `then` res_1 `else` res_2. We use a conditional rather than a traditional disjunction to avoid backtracking during typechecking.

Resource predicates have special syntax to handle the division of their arguments into inputs and outputs. An occurrence of a predicate is written $\alpha(ptr, iargs)(\,oarg\,)$. This is read as the predicate α, applied to a pointer argument *ptr* and a list of other input arguments *iargs*. The output argument *oarg* is highlighted and in a second set of parentheses. Every predicate has exactly one output argument, of type record (with zero or more fields). A *qpred* represents the iterated separating conjunction of predicate instances; it quantifies over integer indices x satisfying a guard *iguard*, and is with input arguments *iargs* and output *oarg*. It represents an instance of α beginning at *ptr*, and repeating every *step* bytes, for as long as the *iguard* is true.

Each resource type has introduction and elimination forms – e.g. $res_1 * res_2$ has pairing and pattern matching proof terms. The standard resource types have the expected rules, and predicate types can be introduced by explicitly folding a predicate definition `fold` *res_term*:*pred*, and unfolded via pattern-matching.

In addition, there are resource operations recording the resource-manipulation steps that inference uses to successfully type a program. If we suppress the book-keeping of checking that input arguments match, calculating indices, and updating output arguments, most of these operations have simple intuitions. `explode` (*res_term*) and `implode` (*res_term*, *tag*) are operations on structs and their members; the first takes an `Owned` $\langle$`struct` *tag*$\rangle$ and turns it into a `Owned` $\langle\tau_i\rangle$ for each of of its members; the second does the inverse. `iterate` (*res_term*, *int*) and `congeal` (*res_term*, *int*) function similarly, but for C's fixed-size arrays, returning a *quantified* `Owned` $\langle\tau\rangle$ instead.

Morally, `break` has type $qpred \to qpred * pred$: it extracts a single predicate from a quantified one, and must return the remainder as well because resource terms are linearly typed; `glue` has type $qpred * pred \to qpred$: it is the inverse to `break`; `split` has type $qpred * iguard \to qpred * qpred$: given a quantified predicate of index-guard $iguard'$, and an *iguard*, if *iguard* implies $iguard'$ then it splits the given quantified predicate into two disjoint parts (one of index-guard *iguard* and the other of $iguard' \wedge \neg\, iguard$); `inj` has type $pred * ptr * step * iargs \to qpred$: it turns a predicate

$$\begin{aligned}
res &::= \mathtt{emp} \mid term \mid pred \mid qpred \mid res_1 * res_2 \mid \exists\, y{:}\beta.\ res' \mid \mathtt{if}\ term\ \mathtt{then}\ res_1\ \mathtt{else}\ res_2 \\
pred &::= \alpha(ptr, iargs)(\,oarg\,) \\
qpred &::= (x; iguard)\{\alpha(ptr + x \times step, iargs)\}(\,oarg\,)
\end{aligned}$$

$$\begin{aligned}
res_term &::= \mathtt{emp} \mid \mathtt{term} \mid pred_term \mid qpred_term \mid \langle res_term_1, res_term_2\rangle \mid \mathtt{pack}\,(oarg, res_term') \\
&\quad\ r \mid \mathtt{fold}\, res_term{:}pred \mid pred_ops \\
pred_ops &::= \mathtt{explode}\,(res_term) \mid \mathtt{implode}\,(res_term, tag) \mid \mathtt{iterate}\,(res_term, int) \\
&\quad\ \mathtt{congeal}\,(res_term, int) \mid \mathtt{break}\,(res_term, term) \mid \mathtt{glue}\,(res_term) \\
&\quad\ \mathtt{inj}\,(res_term, ptr, step, x.\ iargs) \mid \mathtt{split}\,(res_term, iguard)
\end{aligned}$$

Fig. 3. Grammar of Resource Terms

$\alpha(ptr', iargs')$ into a quantified predicate, with $iguard = (x = k)$, where $k = (ptr' - ptr)/step$ and $iargs' = k/x(iargs)$. Because our inference algorithm does not support inferring merging arrays (2.10), there is no inverse to `split` of type $qpred * qpred \longrightarrow qpred$.

4.2 Judgements and Example Rules

The contexts for the rules consist of four parts: (1) C containing the computational variables from the Core program; (2) $\mathcal{L}$ containing purely logical variables mentioned in specifications; (3) Φ, the constraint context, containing a list of (non-quantified) SMT constraints; and (4) $\mathcal{R}$ a *linear* context containing the resources available at that point during type-checking. Assuming a constraint context of only non-quantified constraints is an acceptable simplification, because the elaboration pass can annotate terms with fully-instantiated constraints, whose quantifiers were either supplied by lemmas, annotations or default instantiation.

We focus on the judgements for typing resource terms and memory actions. The judgement $C; \mathcal{L}; \Phi; \mathcal{R} \vdash res\_term \Rightarrow res$ should be read as "under a context of computational variables C, logical variables $\mathcal{L}$, constraints Φ and resources $\mathcal{R}$, the resource term *res_term* synthesises resource type *res*" (the highlighting shows the part of the judgement with an output mode). The judgement $C; \mathcal{L}; \Phi; \mathcal{R} \vdash res\_term \Leftarrow res$ reads similarly, replacing 'synthesises' with 'checks against'.

We need both judgements because variables, folding, and predicate operations are naturally typed as synthesising rules, whereas constraints, packing existentials, and conditional resources require checking. Furthermore, as we shall see soon, memory actions require a synthesising judgement (to obtain and manipulate the output argument of $\mathtt{Owned}\,\langle\tau\rangle$), whereas top-level values require checking judgements.

$$\frac{\mathtt{smt}\,(\Phi \Rightarrow term) \qquad C; \mathcal{L}; \Phi; \mathcal{R} \vdash res\_term \Leftarrow res_1}{C; \mathcal{L}; \Phi; \mathcal{R} \vdash res\_term \Leftarrow \mathtt{if}\ term\ \mathtt{then}\ res_1\ \mathtt{else}\ res_2}$$

$$\frac{\begin{array}{l} pred \equiv \alpha(ptr, \overline{iarg_i}^{\,i})(\,oarg\,) \\ \alpha \equiv x_p{:}\mathtt{pointer},\ \overline{x_i{:}\beta_i}^{\,i},\ y{:}\mathtt{record}\,\overline{tag_j{:}\beta'_j}^{\,j} \mapsto res \in \mathtt{Globals} \\ C; \mathcal{L}; \Phi; \mathcal{R} \vdash res\_term \Leftarrow [oarg/y, [\,\overline{iarg_i/x_i}^{\,i}\,], ptr/x_p](res) \end{array}}{C; \mathcal{L}; \Phi; \mathcal{R} \vdash \mathtt{fold}\, res\_term{:}pred \Rightarrow pred}$$

On the left is one of two rules for checking a conditional resource. Thanks to the ordered disjunction, the rule is simple: if the SMT solver can statically prove *term*, then check the resource term against the res_1. The converse (omitted) checks against res_2 if the SMT solver can prove the negation of the condition; if neither is provable, the rules try to synthesise an under-determined conditional resource (the only way this is possible is if *res_term* is a variable of an SMT-equivalent type). The rule for folding predicates shown is simplified for presentation (omitting only the type checking of the all the predicate arguments, and the exclusion of the

$\texttt{Owned}\,\langle\tau\rangle$ predicate because it cannot be folded). The first line simply looks up the types of the arguments based on the predicate name, and the "body" *res* of the predicate. The second checks *res_term* against the *res* with its arguments (supplied by the fold term) substituted in.

$$\frac{ret \equiv \Sigma\, y_p{:}\texttt{pointer}.\ term \wedge (y_p \overset{\texttt{false}}{\mapsto_\tau} \texttt{default}\,\beta_\tau) * \mathtt{I}}{\mathcal{C};\mathcal{L};\Phi;\cdot \vdash \texttt{create}\,(pval,\tau) \Rightarrow ret}$$

$$\frac{\begin{array}{l}\mathcal{C};\mathcal{L};\Phi;\mathcal{R} \vdash res\_term \Rightarrow term \overset{init}{\mapsto_\tau} pval_1 \\ \texttt{smt}\,(\Phi \Rightarrow (term = pval_0) \wedge (init = \texttt{true})) \\ ret \equiv \Sigma\, y{:}\beta_\tau.\ y = pval_1 \wedge (pval_0 \overset{\texttt{true}}{\mapsto_\tau} pval_1) * \mathtt{I}\end{array}}{\mathcal{C};\mathcal{L};\Phi;\mathcal{R} \vdash \texttt{load}\,(\tau, pval_0, \_, res\_term) \Rightarrow ret}$$

$$\frac{\begin{array}{l}\mathcal{C};\mathcal{L};\Phi;\mathcal{R} \vdash res\_term \Rightarrow term \overset{-}{\mapsto_\tau} \_ \\ \texttt{smt}\,(\Phi \Rightarrow term = pval_0) \\ ret \equiv \Sigma\,\_{:}\texttt{unit}.\ (pval_0 \overset{\texttt{true}}{\mapsto_\tau} pval_1) * \mathtt{I}\end{array}}{\mathcal{C};\mathcal{L};\Phi;\mathcal{R} \vdash \texttt{store}\,(\_, \tau, pval_0, pval_1, \_, res\_term) \Rightarrow ret}$$

$$\frac{\begin{array}{l}\mathcal{C};\mathcal{L};\Phi;\mathcal{R} \vdash res\_term \Rightarrow term \overset{-}{\mapsto_\tau} \_ \\ \texttt{smt}\,(\Phi \Rightarrow term = pval)\end{array}}{\mathcal{C};\mathcal{L};\Phi;\mathcal{R} \vdash \texttt{kill}\,(\texttt{static}\,\tau, pval, res\_term) \Rightarrow \Sigma\,\_{:}\texttt{unit}.\ \mathtt{I}}$$

The rules for typing memory actions are also simplified for presentation. Allocating memory with `create` takes an alignment *pval* and a C-type τ and synthesises a return type *ret* representing: a newly created pointer (y_p), a constraint (*term*) about alignment and representability (omitted), and the resource itself (printed in more familiar $\mapsto$ notation). In addition to the *value* output of the $\texttt{Owned}\,\langle\tau\rangle$ resource (in this case, an unconstrained default value of the appropriate type, $\texttt{default}\,\beta_\tau$), the formalisation includes an initialisedness status *init* (in this case, `false`). Loading from a memory location requires a correctly typed resource term, and that its output argument's initialisedness status *init* is `true`. It returns the pointed-to value and the same permission it consumed. Storing to a memory location requires a points-to permission, but without any constraints on initialisedness. It returns the permission, with initialisedness status `true` and updated value. De-allocating memory with `kill` is the converse of allocating memory: a resource term is required, but not returned.

4.3 Soundness

Theorem 4.1 (Progress and type preservation for resource terms). *For all closed resource terms (res_term) which type check or synthesise ($\cdot;\cdot;\Phi;\underline{\mathcal{R}} \vdash res\_term \Leftarrow res$) and all well-typed heaps ($\Phi \vdash h \Leftarrow \underline{\mathcal{R}}$) there exists a resource value (res_val), context ($\underline{\mathcal{R}'}$) and heap (h'), such that: the value is well-typed ($\cdot;\cdot;\Phi;\underline{\mathcal{R}'} \vdash res\_val \Leftarrow res$); the heap is well-typed ($\Phi \vdash h' \Leftarrow \underline{\mathcal{R}'}$); for all frame-heaps ($f$), the resource term reduces to the resource value without affecting the frame-heap ($\langle h+f; res\_term\rangle \Downarrow \langle h'+f; res\_val\rangle$).*

Proof: §B8.6 of the online materials.

Theorem 4.2 (Progress for the annotated and let-normalised Core). *If a top-level expression (texpr) is well-typed ($\cdot;\cdot;\Phi;\underline{\mathcal{R}} \vdash texpr \Leftarrow ret$) and all computational patterns in it are exhaustive, then either it is a value (tval), or it is unreachable, or for all heaps (h), if the heap is well-typed ($\Phi \vdash h \Leftarrow \underline{\mathcal{R}}$) then there exists another heap (h') and expression ($texpr'$) which is stepped to ($\langle h; texpr\rangle \longrightarrow \langle h'; texpr'\rangle$) in the operational semantics.*

Proof: §B9.6 of the online materials.

Theorem 4.3 (Type preservation for the annotated and let-normalised Core). *For all closed and well-typed top-level expressions ($\cdot;\cdot;\Phi;\underline{\mathcal{R}} \vdash texpr \Leftarrow ret$), well typed heaps ($\Phi \vdash h \Leftarrow \underline{\mathcal{R}}$), frame-heaps (f), new heaps (heap), and new top-level expressions ($texpr'$), which are connected by a step in the operational semantics ($\langle h+f; texpr\rangle \longrightarrow \langle heap; texpr'\rangle$), if all top-level functions are annotated correctly, there exists a constraint context (Φ'), sub-heap (h'), and resource context ($\underline{\mathcal{R}'}$),* *such that the constraint context is Φ extended, the frame is unaffected (heap = $h' + f$), the sub-heap is well-typed ($\Phi' \vdash h' \Leftarrow \underline{\mathcal{R}'}$), and the top-level expression too ($\cdot; \cdot; \Phi'; \underline{\mathcal{R}'} \vdash texpr' \Leftarrow ret$).*

Proof: §B10.2 of the online materials.

5 COMPARISON

The related work has explored many approaches to C code verification; §6 gives an overview. To put CN into context, we compare it with three state-of-the-art tools: VeriFast, RefinedC, and Frama-C.

Semantics and TCB. VeriFast [Jacobs et al. 2011] is a verification tool for C with similar design to CN, based on separation logic. VeriFast uses an ad hoc C semantics, rather further from ISO C, e.g. (unlike CN) treating uninitialised memory as stable and (like CN currently) not taking pointer provenance into account in its memory model. Frama-C [Baudin et al. 2021] is a framework for analysing C programs. Frama-C translates programs into CIL [Necula et al. 2002] for analysis; its tool suite includes WP, a Hoare Logic verifier based on weakest preconditions. WP uses a custom semantics for CIL, parametric in the memory model; users can choose between them based on a trade-off between performance and support for pointer manipulation. RefinedC [Sammler et al. 2021] is a C type system with separation logic and refinement types. It handles C by first translating into a kernel language, *Caesium*, using Cerberus' frontend (but not Core elaboration); Caesium has an ad hoc semantics that does not handle "lifetimes of block-scoped variables" [Sammler et al. 2021] or weak sequencing of memory actions in C (CN handles the former but not the latter); in contrast to CN's fully concrete memory model, RefinedC integrates VIP [Lepigre et al. 2022]. Unlike CN, Frama-C, and VeriFast, RefinedC is implemented inside a theorem prover, in Coq above Iris, and so the TCB is the C semantics and Coq itself, but not typing rules or automation [Sammler et al. 2021]. CN builds *directly* on Cerberus's (elaboration) semantics and benefits from its validation; currently, however, CN does not handle the weak sequencing semantics and uses an overly simple fully concrete memory model. Besides the Cerberus and CN code, CN's TCB includes Z3.

Specification language and expressivity. VeriFast's annotation language is the closest to CN's: it distinguishes ownership assertions from pure assertions in a similar way to CN and provides languages for user-defined separation logic predicates and logical functions; VeriFast supports some features CN currently does not, such as recursive logical functions, as well as automatic lemma application and predicate un/packing. RefinedC is much more expressive than CN or VeriFast: being based on Iris, it has built-in support for higher-order predicates, unrestricted logical quantifiers, and arbitrary Coq terms inside types; these cannot generally be handled fully automatically; users can increase automation by extending its reasoning rules or using custom tactics. Frama-C has a function specification language with C-like syntax that is superficially similar to CN's; however Frama-C is based on Hoare Logic. Frama-C supports a number of features that CN could benefit from: it supports ghost variables, updated via ghost code, it allows multiple specifications for functions ("behaviours"), and it can generate runtime checks from specifications. Compared to these tools, a special feature of CN is the restrictions it imposes on specifications, such as our resource-let-binding syntax and typing rules that prevent non-linear integer arithmetic (except via uninterpreted functions), to ensure predictable type checking.

Annotation overhead and performance. To compare annotation overhead and verification performance, we verify a smaller case study in each tool: pKVM's "early allocator", a simple allocator without memory reclamation, which pKVM uses during initialisation (before using the buddy allocator of §3). The source code consists of 96 lines of C code and macro definitions, and can be found at https://github.com/rems-project/pKVM-early-allocator-case-study. The allocator's main functions are `hyp_early_alloc_init`, which initialises the allocator; `hyp_early_alloc_nr_pages`, which calculates how many pages have already been allocated; and `hyp_early_alloc_page`, which returns a `void` pointer to a freshly allocated page, zeroed using `clear_page`. We have also included `hyp_early_alloc_contig` from a newer version of the early allocator, which takes an `unsigned int nr_pages` and allocates several adjacent pages; moreover, since `clear_page` is originally written in assembly, we have added a simple C implementation.

We formalise the early allocator in each of the four systems, specifying function pre- and post-conditions that guarantee safe execution, correct ownership transfer, and certain properties of the resulting values. For RefinedC we base the formalisation on an existing one [Lepigre and Sammler 2022; Lepigre et al. 2022]. Since one of the types (for regions of zero'ed memory) does not currently have good RefinedC support, we skip the RefinedC proof of `clear_page` and a loop in `hyp_early_alloc_contig` (accounting for this in calculating the formalisation overhead). Since Frama-C is based on Hoare Logic rather than separation logic, it does not easily support the ownership specifications used for the other tools; our Frama-C specifications therefore are weaker than those for the other tools. The table on the right shows the formalisation overheads and verification running times. "Instr." counts intra-function instrumentation: pack and unpack statements for VeriFast and CN, and pointer provenance hints for RefinedC. The Frama-C formalisation is concise, but not directly comparable (see above). Among the separation logic tools, VeriFast has the most concise specifications and smallest overhead, indicating room for improvement for CN. CN, Frama-C, and VeriFast, all verify the code fairly quickly; in the case of Frama-C the solver times out for one proof goal after 10s; in the statistics we use a 1s limit (with the same one goal timing out). RefinedC is implemented inside Coq, so higher running times are expected. We run the tools inside a Ubuntu 22.04 VirtualBox with 8GB ram restricted to two cores, on a 2GHz Quad-Core Intel Core i5 machine.

tool	*spec.*	*instr.*	*defs.*	*proof*	*total*	*time*
CN	47	7	17	0	0.74x	0.47s
Frama-C	38	0	0	0	0.40x	3.46s
RefinedC	50	2	5	8	0.77x	16.72s
VeriFast	38	5	14	3	0.62x	0.05s

spec./instr./defs./proof are line counts, x: overhead factor, s: seconds

6 RELATED WORK

Xi introduced the idea of combining solver-resolved logical constraints and typechecking in his work on Dependent ML [Xi 2007], which combined much of the expressiveness of dependent type theory with a lightweight, "proof-free" style of programming. DML permitted a very rich language of logical constraints, and so did not have decidable type inference. Rondon et al. [2008] introduced *liquid types*, which can be seen as a natural restriction of DML supporting decidable type inference. The two key ideas behind liquid types are (1) to restrict the logical constraints to be quantifier-free, and (2) to ensure that the quantifiers occurring in types track program values (function arguments for universal quantifiers, function return values for existentials). The second restriction permits quantifier instantiations to be resolved by looking at the arguments to a function, ensuring that all the logical generated entailments remain quantifier-free and hence SMT-solvable. CN adopts essentially the liquid types methodology to ensure that refinement checking is decidable, with some additions for quantified variables in resource types and to support a restricted form of quantified constraints with default instantiation and manual instantiation by the user.

Using substructural types to track mutable data is a very old idea, dating back to Reynolds' *syntactic control of interference* [Reynolds 1978] and Girard's *linear logic* [Girard 1987]. Reynolds (with Peter O'Hearn) also invented *separation logic*, the most successful extension of Hoare logic for reasoning about heap-manipulating programs. CN's resource types are a hybrid of separation logic and linear types. Unlike a pure linear type system, all program values in CN are nonlinear. Instead (inspired by L3 [Ahmed et al. 2007]), resource ownership is decoupled from individual values and managed linearly. The added flexibility of this approach simplifies verifying complex ownership transfers which do not follow the structure of the data (as in the buddy allocator).

CN's resource language must be a subset of separation logic, since our liquid types discipline requires decidable logical constraints. However, expressing the ownership of a linked list requires existential quantifiers. CN's approach – distinguishing between input and output positions to ensure existential constraints – is not novel, and is used by several earlier systems like VeriFast [Jacobs et al. 2011] and Gillian [Maksimovic et al. 2021]. However, both of these systems permit writing assertions which fall outside the well-moded fragment, and our realisation that the moding problem can be formulated as a variable scoping problem is novel. Furthermore, Brotherston et al. [2016] show that this fragment can be model-checked in polynomial time, which suggests that this style of assertion should also be amenable to *runtime* checking (though we leave this to future work).

The system closest in design to CN is VeriFast [Jacobs et al. 2011]. VeriFast is also designed to support predictable, decidable verification of separation logic proofs, though it is structured more like a program logic rather than a type system. The biggest design differences are that VeriFast uses an ad hoc C semantics and that it (like ATS [Cui et al. 2005], discussed below) treats the array predicate as an inductive assertion, and so random access to array elements requires lemmas. VeriFast also has features that CN does not yet have, such as user-defined inductive types and some automation of opens and closes, which are not substantial design differences but are very important to the user experience.

Since random array access is important for our target applications, we needed to support richer automation for array accesses. We went through several iterations of our automation before settling on our current scheme. The traditional SMT theory of arrays is insufficiently expressive, and even extensions like the scheme of [Bradley et al. 2006] were still insufficiently expressive for the invariants required for our buddy allocator verification. The Viper framework [Müller et al. 2017] (used in the Prusti tool [Astrauskas et al. 2022]) has rich support for modelling arrays as iterated separation conjunctions [Müller et al. 2016]. Our implementation of the Viper array inference algorithm did not perform well, and we adopted a modified scheme (§2.10).

Also close in design to CN is ATS [Cui et al. 2005]. ATS extends a full functional programming language (including higher-order functions and polymorphism) with support for linear resource types. It can be seen as something like a higher-order, polymorphic version of kernel CN: in particular, all linear types are tracked by explicit proof terms, which occur in the source, but are not relevant at runtime. This makes it possible to express many rich ownership transformations, at the price of even heavier annotations than CN. It is also its own language, with C as a compiler target.

Smallfoot [Berdine et al. 2005] initiated a line of work on developing tools to automatically derive proofs for fixed, decidable fragments of separation logic, later reaching widespread production usage with Infer [Calcagno and Distefano 2011]. These tools require very little annotation, but do not prove strong properties – bug-finders rather than functional-correctness verification tools.

At the other end of the spectrum, systems like the Verified Software Toolchain [Appel et al. 2014], Iris [Jung et al. 2018], and the C semantics of Krebbers [2015] build complete toolchains for doing interactive separation logic proofs in the Coq proof assistant. In these systems the specification language is a maximal one (with assertions as shallow embeddings in Coq), and proofs are mostly manual (a 10-to-1 ratio of proof to code is typical for nontrivial Iris developments). VST is built atop a verified compiler, CompCert [Leroy 2006], and specialises its logic to the CompCert dialect of C, rather than ISO (or clang, or gcc) C. Iris is generic over the programming language.

In between these two extremes are Boogie [Barnett et al. 2005], F* [Swamy et al. 2016] and similar tools, as well as RefinedC [Sammler et al. 2021]. RefinedC is a refinement type system based on separation logic. It instantiates Iris for C and provides significant proof automation using its Lithium language; users can extend automation with new rules and provide custom tactics. RefinedC uses the Cerberus front-end, but then its own ad hoc semantics, not the Cerberus elaboration. Boogie and F* make heavy use of SMT-based automation to discharge proof obligations. They both freely use Z3's support for quantifiers, and as such aim at tactic-based proof automation rather than reliable inference. Boogie implements a program logic for a custom low-level language, and F* a dependent type theory, but both support generating C code from verified programs.

More distant from CN are tools like Frama-C [Baudin et al. 2021], which implements (essentially) pure Hoare logic (rather than separation logic) for C, with support for manually discharging proofs that automation cannot solve. (Boogie used to be in this category, but has recently grown support for linear types, making it more similar to CN.)

There have been many verification projects of low-level systems code, using an equally wide array of approaches. Many involve substantial manual work in a theorem prover, e.g. the verification of the seL4 [Klein et al. 2010] microkernel in Isabelle by refinement, or the verification of the CertiKOS [Gu et al. 2016] and SeKVM [Li et al. 2021] kernels in Coq via a stack of smaller refinement proofs, or the BedRock hypervisor [Malecha et al. 2022], being verified by instantiating the Iris separation logic with weakest precondition rules for a fragment of C++, in Coq. More similar to CN are more automated approaches, e.g. verification of the Hyper-V hypervisor [Leinenbach and Santen 2009] and Ironclad environment [Hawblitzel et al. 2014] using the Boogie system. These projects have typically built a clean-slate implementation (e.g. seL4, CertiKOS), or even extracted it from their proof environment (e.g. Ironclad).

All these systems have their own limitations, making direct comparison difficult; it would be useful for the community to develop a common set of C functional correctness verification examples, with a range of C feature and programming idiom usage. As a sanity check, we proved in CN some example problems from VST and VeriFast. These examples are provided as gentle introductions that work smoothly in those tools, and sometimes require more manual work (lemmas, etc.) in CN. Conversely, we believe the low-level systems-code idioms of the pKVM buddy allocator would make it challenging to verify in many other systems.

Overall, there are almost as many different approaches as there are verified systems, indicating that the community still does not have a good sense of what the tradeoffs in this space are. CN adds a promising and previously unexplored direction, and the successful verification of pKVM's buddy allocator using CN is encouraging. We plan to apply CN to more parts of pKVM, study whether its design scales, and discover and add whatever extensions are required for handling larger verifications of such conventional production systems code.

DATA ACCESS

Data accompanying this publication can be accessed at [Pulte et al. 2022].

ACKNOWLEDGMENTS

We thank Jean Pichon-Pharabod for valuable discussions. We thank the anonymous reviewers for their helpful feedback. Finally, we thank Ben Laurie, Sarah de Haas, Hongseok Kim, the pKVM development team, and the Android Security and Android Platform teams for their support. This work was supported in part by a European Research Council (ERC) Consolidator Grant for the project "TypeFoundry", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 101002277), in part by a European Research Council (ERC) Advanced Grant "ELVER" under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 789108), in part by an EPSRC Doctoral Training studentship, and in part by funding from Google Research.

REFERENCES

Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L^3: A Linear Language with Locations. *Fundam. Informaticae* 77, 4 (2007), 397–449.

Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. 2013. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis - Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers*. 1–18. https://doi.org/10.1007/978-3-319-12466-7_1

Android Open Source. 2022. Android 13 Release Notes. https://source.android.com/docs/setup/about/android-13-release. [Online; accessed 11-November-2022].

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. CUP. https://doi.org/10.1017/CBO9781107256552

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 *NFM (Lecture Notes in Computer Science, Vol. 13260)*. Springer, 88–108.

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO (Lecture Notes in Computer Science, Vol. 4111)*. Springer, 364–387.

Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. 2021. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM* 64, 8 (2021), 56–68.

Christoph Baumann, Mats Näslund, Christian Gehrmann, Oliver Schwarz, and Hans Thorsen. 2016. A high assurance virtualization platform for ARMv8. In *European Conference on Networks and Communications, EuCNC 2016, Athens, Greece, June 27-30, 2016*. 210–214. https://doi.org/10.1109/EuCNC.2016.7561034

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO (Lecture Notes in Computer Science, Vol. 4111)*. Springer, 115–137.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2006. What's Decidable About Arrays?. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3855)*, E. Allen Emerson and Kedar S. Namjoshi (Eds.). Springer, 427–442. https://doi.org/10.1007/11609773_28

James Brotherston, Nikos Gorogiannis, Max I. Kanovich, and Reuben Rowe. 2016. Model checking for symbolic-heap separation logic with inductive predicates. In *POPL*. ACM, 84–96.

Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (Lecture Notes in Computer Science, Vol. 6617)*. Springer, 459–465.

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

Sa Cui, Kevin Donnelly, and Hongwei Xi. 2005. ATS: A Language That Combines Programming with Theorem Proving. In *FroCoS (Lecture Notes in Computer Science, Vol. 3717)*. Springer, 310–320.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

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.

Jake Edge. 2020. KVM for Android. Linux Weekly News, LWNarticle](https://lwn.net/Articles/836693/. Accessed 2022-07-07.

Anthony C. J. Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. 2017. Verified compilation of CakeML to multiple machine-code targets. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. 125–137. https://doi.org/10.1145/3018610.3018621

Dan Frumin, Léon Gondelman, and Robbert Krebbers. 2019. Semi-automated Reasoning About Non-determinism in C Expressions. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 60–87. https://doi.org/10.1007/978-3-030-17184-1_3

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. 2016. Provably secure memory isolation for Linux on ARM. *Journal of Computer Security* 24, 6 (2016), 793–837. https://doi.org/10.3233/JCS-160558

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 165–181. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

Gernot Heiser, Gerwin Klein, and June Andronick. 2020. seL4 in Australia: from research to real-world trustworthy systems. *Commun. ACM* 63, 4 (2020), 72–75. https://doi.org/10.1145/3378426

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

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.

Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM TOCS* 32, 1 (Feb. 2014), 2:1–2:70. https://doi.org/10.1145/2560537

Robbert Krebbers. 2015. *The C Standard Formalized in Coq.* Ph. D. Dissertation. Radboud University Nijmegen.

Robbert Krebbers. 2016. A Formal C Memory Model for Separation Logic. *J. Autom. Reason.* 57, 4 (2016), 319–387. https://doi.org/10.1007/s10817-016-9369-1

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings.* 806–809. https://doi.org/10.1007/978-3-642-05089-3_51

Rodolphe Lepigre and Michael Sammler. 2022. RefinedC formalisation of the early allocator. https://gitlab.mpi-sws.org/iris/refinedc/-/tree/master/linux/pkvm. [Online; accessed 24-October-2022].

Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. 2022. VIP: verifying real-world C idioms with integer-pointer casts. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–32. https://doi.org/10.1145/3498681

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. ACM, 42–54.

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1782–1799. https://doi.org/10.1109/SP40001.2021.00049

Petar Maksimovic, José Fragoso Santos, Sacha-Élie Ayoun, and Philippa Gardner. 2021. Gillian: A Multi-Language Platform for Unified Symbolic Analysis. *CoRR* abs/2105.14769 (2021).

Gregory Malecha, Gordon Stewart, Frantisek Farka, Jasper Haag, and Yoichi Hirai. 2022. Developing With Formal Methods at BedRock Systems, Inc. *IEEE Security & Privacy* 20, 3 (May 2022), 33–42. https://doi.org/10.1109/MSEC.2022.3158196

Nicholas D. Matsakis and Felix S. Klock II. 2014. The Rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, Michael Feldman and S. Tucker Taft (Eds.). ACM, 103–104. https://doi.org/10.1145/2663171.2663188

Paul-André Melliès and Noam Zeilberger. 2015. Functors are Type Refinement Systems. 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, 3–16. https://doi.org/10.1145/2676726.2676970

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.

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

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 405–425. https://doi.org/10.1007/978-3-319-41528-4_22

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2017. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 50. IOS Press, 104–125.

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer, 213–228. https://doi.org/10.1007/3-540-45937-5_16

Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *J. Funct. Program.* 31 (2021), e25. https://doi.org/10.1017/S095679682100023X

Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2022. Data for "CN: Verifying Systems C Code with Separation-Logic Refinement Types". https://doi.org/10.5281/zenodo.7320414

John C. Reynolds. 1978. Syntactic Control of Interference. In *POPL*. ACM Press, 39–46.

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

John C. Reynolds. 2008. An Introduction to Separation Logic (Preliminary Draft). "https://www.cs.cmu.edu/~jcr/copenhagen08.pdf". "[Online; accessed 4-July-2022]".

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *PLDI*. ACM, 159–169.

Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: verification of machine code against authoritative ISA semantics. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 825–840. https://doi.org/10.1145/3519939.3523434

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

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2016. A new verified compiler backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 60–73. https://doi.org/10.1145/2951913.2951924

Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. 2021. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 866–881. https://doi.org/10.1145/3477132.3483560

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

Received 2022-07-07; accepted 2022-11-07