UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

Translating CN Separation Logic Lemmas to the Rocq Proof Assistant

Amy Yin

Lucy Cavendish College

June 2025

Submitted in partial fulfillment of the requirements for the
Master of Philosophy in Advanced Computer Science

Total page count: 58

Main chapters (excluding front-matter, references and appendix): 48 pages (pp 7–54)

Main chapters word count: 13868

Methodology used to generate that word count:

Overleaf wordcount button.

All text occurring before Chapter 1 is surrounded by `%TC:ignore` and `%TC:endignore`.

Declaration

I, Amy Yin of Lucy Cavendish College, being a candidate for the Master of Philosophy in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. I am content for my report to be made available to the students and staff of the University.

Signed: Amy Yin

Date: June 10th, 2025.

Abstract

CN is a separation-logic refinement type system and verification tool for C code, equipped with an SMT backend. This project extends CN's existing mechanism for exporting lemmas falling outside a decidable SMT fragment to the Rocq proof assistant, with a focus on making proofs concerning resource ownership possible through the Iris separation logic framework.

Acknowledgements

This project would not have been possible without the wonderful support of Christopher Pulte, Neel Krishnaswami, Dhruv, Anders, Meven, Arthur, Ines, Kuba, Thomas, and Thomas.

Contents

Chapter 1

Introduction

Much systems software exists in the world, and much of it is written in the C programming language. Mathematically verifying the correctness of such systems against a formal specification to ensure useful properties such as memory safety is important because they may be employed in critical infrastructure such as nuclear submarine control panels and high-speed trading operations.

Decades of research in programming languages and verification have been dedicated towards the development of verified systems software such as compilers [20] [10], operating systems [11], hypervisors [12] [36], and more. However, a large gulf remains between such efforts and the industrial-grade C code shipped by real C programmers out in the wild. The nature of this gulf may be summarised through the two following factors.

I. For one, C's notoriously complicated semantics often compels researchers to make simplifying assumptions that restrict the language until it no longer resembles what programmers in the industry are using.

II. Existing verification tools tend to inhabit one out of two extremes: They are either highly manual and require expertise in a proof assistant to use, or are automated through SMT techniques to the extent that verification becomes unpredictable and errors difficult to diagnose.

The CN verification tool [30] is a recent development based on separation logic that tackles the two difficulties listed above. Firstly, CN is built upon Cerberus [23], a semantics for a large fragment of ISO C that is both well-validated and realistic. Secondly, CN's verification process is two-fold: it is equipped with an SMT backend that automatically verifies logical constraints within a decidable SMT fragment, and remaining constraints can be manually stated as specification-level lemmas. These lemmas can then be exported to the Rocq interactive proof assistant, where further manual proof can be carried out. However, CN's existing implementation of this lemma export mechanism is incomplete and lacks support for in-demand features such as specifications using recursive functions and

lemmas involving ownership of memory. The aim of this project is to add support for these missing features in the CN codebase, as well as refactoring the existing implementation of the lemma export mechanism and using our new implementation to prove a number of lemmas needed by real CN users.

1.1 Contributions

This project makes the following contributions:

1. We present translations from CN separation logic lemmas to the Rocq proof assistant, using the Iris separation logic framework. We implement this translation in OCaml, as an extension to the CN codebase.
2. We design a Rocq library to accompany our translation, which contains an instantiation of Iris, definitions that facilitate reasoning about resource ownership, and a number of helpful lemmas about their behaviour.
3. We refactor the existing CN codebase by introducing an intermediate AST for our translation.
4. We provide Rocq proofs for a number of lemmas generated by our implementation based on realistic examples of CN usage, discovering an error in an example employed by the official CN documentation in the process.

1.2 Structure of This Report

In Chapter 2 we present the background knowledge required to understand the rest of this report, including an overview of CN's theoretical foundations in separation logic as well as CN itself.

In Chapter 3 we discuss related work on the verification of C, using separation logic specifications or otherwise.

We then detail our implementation in Chapter 4, and evaluate it by using it to generate and prove lemmas for three case studies in Chapter 5. We summarise our work and discuss future work on this project in Chapter 6.

Chapter 2

Background

In this chapter we introduce separation logic, which acts as the theoretical basis for our work. We then introduce the CN verification tool, the Rocq proof assistant, and the Iris separation logic framework within Rocq.

2.1 Separation Logic

Separation logic is an extension of Hoare logic that facilitates reasoning about imperative programs with explicit memory management. The CN project uses separation logic to verify the behaviour of C programs, and in this section we introduce a list of relevant definitions for reference in future chapters.

2.1.1 Hoare Logic

Hoare logic is a system for reasoning about the behaviour of programs in terms of its *pre-* and *post-conditions* on program variables. Its most important feature is the *Hoare triple*, written

$$\{P\}\ C\ \{Q\}$$

where P is a pre-condition, C is a program (or "command"), and Q is a post-condition. P and Q are predicate logic formulae, also called *assertions*, that contain conditions on the program variables occurring in C. The triple is read as "if P is true, then Q is true upon execution of C" [13]. For example, the following triple has the pre-condition that the program variable x has value 1, and post-condition that x has value 2. The program itself uses the assignment command to set the value at x to that of $x + 1$.

$$\{x = 1\}\ x := x + 1\ \{x = 2\}$$

We can easily verify this triple is true. For contrast, see below for an invalid triple:

$$\{x = 1\}\ x := x + 1\ \{x = 100\}$$

Hoare logic comes with a proof system for deriving valid triples for commands written in a simple imperative language. This system comprises of a number of axioms and inference rules. For example, the axiom schema of assignment is defined as

$$\frac{}{\{P[f/x]\}\ x := f\ \{P\}}\ \text{ASSIGNMENT}$$

where $P[f/x]$ denotes the substitution of a pure expression f for all free instances of a variable x in the assertion P. This axiom schema states if the postcondition $\{P\}$ is true after the assignment $x := f$, the precondition $\{P[f/x]\}$ must be true before the assignment is performed [13].

For an example of an inference rule, consider the rule of composition

$$\frac{\{P\}\ Q_1\ \{R\} \qquad \{R\}\ Q_2\ \{S\}}{\{P\}\ Q_1; Q_2\ \{S\}}\ \text{COMPOSITION}$$

where $Q_1; Q_2$ denotes sequential composition of the commands Q_1 and Q_2, i.e. executing them one after the other. The rule states that if the post-condition of Q_1 is the same as the pre-condition Q_2, we may derive that the pre-condition of $Q_1; Q_2$ is that of Q_1, and its post-condition is that of Q_2.

A number of such axioms and inference rules exist for other language constructs one might find in imperative programs, such as while loops, if-then-else statements, and more. While Hoare triples are not directly used in CN's verification process, we will use them to motivate the correspondence between CN lemmas and separation logic statements in a later section.

2.1.2 Resource Ownership

Hoare logic is not well-suited for reasoning about programs that involve pointer and heap manipulations because it does not contain the constructs necessary to represent them, rendering it non-ideal for reasoning about a large subset of C programs. In contrast, separation logic is an extension of Hoare logic equipped with the appropriate constructs: it introduces a mathematical definition of heaps, and enables reasoning about the *ownership* of heap cells by defining a notion of *separation* between heaps, through which it becomes possible to model the behaviour of pointers.

In separation logic, heaps are defined as finite partial functions

$$L \rightharpoonup V$$

where L is a memory address, and V is the value stored at that address. The type of L varies depending on what addresses and allocations are possible in the language being modelled, and is typically taken to be the non-negative integers because this means there will be infinitely many sequences of consecutive integers of arbitrary length available, such that memory allocation always succeeds. Similarly, the type of V depends on the types of values available in the programming language being verified [27]. We write *emp* for the empty heap.

We now introduce a new assertion called the *points-to assertion*, written as

$$t_1 \mapsto t_2$$

This assertion states that

- The heap location t_1 maps to the value t_2, and
- We have ownership of the heap location t_1.

Ownership is a loosely-articulated notion derived from the *sharing interpretation* of computational resources [28]. O'Hearn states that ownership is synonymous with "the right to dereference", such that *emp* should be read as "I don't have permission to dereference any heap", or "I own nothing" [26].

2.1.3 Separating Conjunction and the Magic Wand

To motivate the separating conjunction operator $\star$ and the "magic wand" operator $-\!\!\star$, we consider the following Hoare logic inference rule known as the rule of constancy [32]:

$$\frac{\{P\}\ C\ \{Q\}}{\{P \wedge R\}\ C\ \{Q \wedge R\}}\ \text{CONSTANCY}$$

This rule states that if $\{P\}\ C\ \{Q\}$ is a valid Hoare triple, then its pre- and post-conditions may be extended by taking their conjunction with another assertion R, under the assumption that C does not modify any free variables in R.

However, this rule becomes unsound when separation logic-style assertions about heaps such points-to predicates are permitted, because of counter-examples such as the following:

$$\frac{\{x \mapsto 0\}\ x := 42\ \{x \mapsto 42\}}{\{x \mapsto 0 \wedge y \mapsto 1\}\ x := 42\ \{x \mapsto 42 \wedge y \mapsto 1\}}$$

Here, the conclusion is not true if x and y are aliases, in which case the postcondition will be $\{x \mapsto 42 \wedge y \mapsto 42\}$, rather than $\{x \mapsto 42 \wedge y \mapsto 1\}$ as implied by the rule of constancy.

The separating conjunction operator can be used to describe the heap conditions that must be met for some version of the rule of constancy to hold. We write $h_0 \perp h_1$ to denote that the heaps h_0 and h_1 have disjoint domains, as well as $h_0 \cdot h_1$ to denote their union. Then, the separating conjunction of two assertions

$$P_0 \star P_1$$

is an assertion stating the assertions P_0 and P_1 hold across two disjoint parts of the heap, such that they cannot both access the same memory location. Now, we can express a version of the constancy rule that holds in separation logic, known as the frame rule, which states that a separation logic triple preserves assertions disjoint from its pre-condition:

$$\frac{\{P\}\ C\ \{Q\}}{\{P\ \star\ R\}\ C\ \{Q\ \star\ R\}}\ \text{FRAME}$$

The frame rule allows us to specify those parts of the heap that are not modified by the command C, under the assumption that no free variables in R are modified by C [32].

We now discuss the separating conjunction in more detail. We write $[\![P]\!]$ for the semantics of a separation logic assertion P, which is the set of states that satisfy P, and $[\![P]\!]\ s\ h$ for some store s and heap h to express that P holds for s and h [32]. The semantics of the separating conjunction is

$$\begin{aligned}[\![P_0\ \star\ P_1]\!]\ s\ h \text{ iff } \exists h_0,\ h_1.\ (h_0 \perp h_1) \\ \wedge\ (h_0\ \cdot\ h_1 = h) \\ \wedge\ [\![P_0]\!]\ s\ h_0 \\ \wedge\ [\![P_1]\!]\ s\ h_1.\end{aligned}$$

Meanwhile, the magic wand between two heaps $P_0\ \mathrel{-\!\!\star}\ P_1$ is an assertion stating that if the existing heap is extended with another heap disjoint from it, for which P_0 holds, then P_1 must also hold in the resulting heap. Formally, this is written

$$[\![P_0\ \mathrel{-\!\!\star}\ P_1]\!]\ s\ h \text{ iff } \forall h'.\ (h' \perp h\ \wedge\ [\![P_0]\!]\ s\ h') \rightarrow [\![P_1]\!]\ s\ (h \cdot h').$$

2.2 The CN Specification Language

The CN verification tool uses Cerberus, a realistic semantics for a large fragment of C [23], to implement a type system based on separation logic [24] with the goal of semi-

automatically verifying C code according to user-defined code annotations [30]. These code annotations are separation logic specifications written in the *CN specification language*. To illustrate CN's capabilities, we will employ the running example of an inductively defined integer array. In particular, we discuss how a C array can be described at the specification level using the CN specification language, as well as what a *CN lemma* looks like.

2.2.1 Datatype and Function Definitions

To define an array at the specification-level, we employ mathematical lists where each list element is associated with an owned pointer. To express this in the CN specification language, we must first supply a definition for mathematical lists using a CN-level datatype declaration, indicated by the use of the `datatype` keyword:

```
/*@
datatype List {
  Nil {},
  Cons {i32 Head, datatype List Tail}
}
@*/
```

The definition above is the usual inductive definition of lists used in functional settings, which states a `List` is either the empty list `Nil`, or the result of applying `Cons` on an element `Head` of type `i32` and an existing `List`. CN also supports user-supplied function definitions using by the `function` keyword, through which we can define useful operations on our inductive list such as `Append`, which appends a `List L2` to the end of a `List L1`.

```
/*@
function [rec] (datatype List) Append(datatype List L1, datatype List L2)
{ match L1 {
    Nil {} => {
      L2
    }
    Cons {Head : H, Tail : T}  => {
      Cons {Head: H, Tail: Append(T, L2)}
    }
  }
}
@*/
```

Here, `Append` pattern-matches on the input `List L1`, returning `L2` if `L1` is empty, or recursively calls `Append` on the `Tail` of `L1` and `L2` while `Cons`-ing the `Head` of `L1` onto the result of the recursive call. The `[rec]` flag indicates to CN that `Append` is recursive.

2.2.2 Resource Ownership

We now consider how each element of a `List` can be associated with an owned pointer. To explain how resource ownership works in CN, we first move to a simpler example. Consider the following C program:

```
unsigned int read (unsigned int *p)
    { return *p; }
```

Here, `read` is a C function that takes an unsigned integer pointer as input and returns the value of its pointee. For `read` to be safe when called on some pointer p, we must possess ownership of p, and there must exist some value V_1 at the memory address pointed to by p. The ownership of p should be returned upon execution of the program, such that p still contains some value V_2. We can directly state these specifications as a separation logic triple like the following:

$$\{\ p \mapsto V_1\ \}\ \texttt{read(p)}\ \{\ \exists\ V_2.\ p \mapsto V_2\ \}$$

Then, suppose we also wished to specify that the pointee value at p is unchanged by executing `read`. In the context of our separation logic triple, this entails scoping V_1 across the entirety of the triple such that it is bound in the post-condition, such that we may write the assertion $V_1 = V_2$ in the post-condition like the following.

$$\{\ p \mapsto V_1\ \}\ \texttt{read(p)}\ \{\ \exists\ V_2.\ (p \mapsto V_2) \wedge (V_1 = V_2)\ \}$$

This separation logic assertion corresponds to the following CN annotations for `read`. CN annotations are delimited `/*@ like so @*/`, and the contents of the `requires` and `ensures` clauses correspond directly to the specifications stated above. Submitting this annotated program to CN verifies its behaviour against its annotations.

```
unsigned int read (unsigned int *p)
    /*@ requires take V1 = Owned<unsigned int>(p);
        ensures take V2 = Owned<unsigned int>(p);
            V1 == V2;
    @*/
    { return *p; }
```

In CN, points-to predicate like those appearing in the pre- and post-conditions above are expressed as *resource-take-bindings*:

```
take V = Owned<unsigned int>(p); P
```

Here, the `take` keyword is a CN-specific syntactic restriction that ensures automated solvers can successfully infer ghost variables and existential witnesses. The way this works

is that CN partitions a resource predicate, such as a points-to predicate $p \mapsto V$, into *inputs* and *outputs*, corresponding to the intuition that fixing the inputs to a resource (such as the pointer p in our points-to predicate) should uniquely determine its output (the pointee value V) [30]. In this example, `take` binds `V` to the output of `Owned<unsinged int>(p)`, and denoting the indexing of a points-to predicate by the size of the C-type `unsigned int` as $\mapsto_u$, this example corresponds to the separation logic predicate

$$\forall V.\ (p \mapsto_u V) \mathrel{-\!\!\star} P$$

when it occurs within a `requires` clause, and

$$\exists V.\ (p \mapsto_u V) \star Q$$

when it occurs within an `ensures` clause. The scope of V is Q in both cases.

In addition to `Owned`, CN supports `Block` predicates which are points-to predicates for uninitialised memory, and iterated resources expressed as `each` predicates of the form

$$\texttt{each}(< bty >< id >; t_0)\ \{p(t_1, ..., t_n)\}$$

which corresponds to taking the separating conjunction over each instance of the predicate $p(t_1, ..., t_n)$ for which the guard t_0 is satisfied. More concretely, an `each` predicate that asserts ownership of `n` consecutive unsigned integers at a pointer `p` (i.e. ownership of an array of unsigned integers with base pointer `p`) might look like the following.

```
each (i32 i; 0i32 <= i && i < n)
    { Owned<unsigned int>(array_shift<unsigned int>(p,i)) }
```

Here, `array_shift` is a CN library function that shifts the pointer `p` by `i` positions at the size of the C-type `unsigned int`. Predicates like this are useful when reasoning about the ownership of arrays.

2.2.3 Inductive Resource Predicates

Inductive resource predicates are a separation logic construct used for describing data structures stored on the heap [32]. CN supports reasoning about such data structures by requiring users to supply specification-level inductive predicates, indicated by use of the `predicate` keyword.

Returning to our inductive array example, given the CN-level definition of mathematical lists presented in section 2.2.1, we can now define an inductive resource predicate `Array` which associates each member of the list with an owned pointer:

```
1 /*@
2 predicate (datatype List) Array (pointer p, i32 n) {
```

```
3     if (n == 0i32) {
4       return Nil{};
5     } else {
6       take V = Owned<int>(p);
7       take VS = Array(array_shift<int>(p, 1), n-1i32);
8       return (Cons { Head: V, Tail: VS });
9     }
10  }
11  @*/
```

This predicate states that given a `pointer p` and signed integer `n` representing the size of the array, an empty list should be returned when `n = 0`. Otherwise, ownership is asserted at `p`, and `Array` is recursively called on the result of shifting `p` by the size of one integer while `n` is decremented, such that ownership is asserted for `n` consecutive memory cells of size `int`, and a `List` comprising of the values at each such memory cell is returned. Therefore, an assertion such as

```
take vs = Array(p,n)
```

asserts ownership of `n` consecutive memory cells of size `int`, while enforcing that the values contained in these memory cells is bound to a `List vs`, such that users may reason about C arrays on the specification level.

`Array` corresponds to the following predicate in separation logic. Here, the predicate takes an additional argument: the list L representing the return value of `Array`, such that the separation logic predicate *Array* specifies exactly those mathematical lists where each element is associated with an ownership predicate in the same manner as `Array`. The if-statement in `Array` is then expressed as a disjunction of (separating) conjunctions, and all ownership predicates are existentially quantified [32].

$$
\begin{aligned}
&Array(p, n, L) = \\
&\quad (n = 0 \ \wedge L = Nil \wedge emp) \vee \\
&\quad (n \neq 0 \wedge \exists\, V : int.\ (p \mapsto_{int} V) \star \\
&\qquad (\exists\, VS :\ List.\ (Array(array\_shift_{int}(p, 1), n - 1, VS)) \ \star \\
&\qquad\quad L = Cons(V, VS))))
\end{aligned}
$$

We note that CN requires for guarded cases in resource predicates to be mutually exclusive, which is guaranteed by CN automatically including the negation of all previous guards in each subsequent case, and is reflected in the separation logic predicate given above.

2.2.4 Lemmas

A lemma declaration expresses a logical implication, using the same requires/ensures specification language as function specifications, and allow users to supply pieces of reasoning that cannot be automatically solved by CN's SMT backend. There are several situations where this may be the case. For one, C's external functions, indicated by the C-level keyword `extern`, do not have definitions available for CN to reason about, so assertions involving them are often not solvable by the SMT backend alone. Another more common scenario arises from the fact that CN is designed to only generate logical constraints falling within a decidable SMT fragment, such that any reasoning falling outside this fragment is only supported through the manual invocation of a lemma. This happens whenever proofs by induction are needed, as is the case with `array_lemma`, defined below:

```
/*@
lemma array_lemma (pointer p, i32 n, i32 m)
  requires
    take vs = Array(p,n);
    take ws = Array(array_shift<unsigned int>(p,n),m);
  ensures
    take xs = Array(p,n+m);
    xs == Append(vs,ws);
@*/
```

This lemma states that two `Array`s `vs` and `ws` that inhabit two neighbouring sections of the heap may be viewed as one larger array, and that the elements of this array is equal to the result of `Append`-ing the elements of `vs` to `ws`. Upon declaration, a lemma like `array_lemma` can be manually invoked in a program specification to aid CN's reasoning, and the proof obligation for that lemma is exported into a Rocq file containing its Rocq translation, as well as all definitions available in the CN *global typing context* (CN function definitions, predicates, etc.) that may be used to prove it. Currently, CN only supports the translation of a subset of *pure* lemmas that do not involve resources or recursive functions, so lemmas like `array_lemma` cannot be exported. The primary aim of this project is to extend translation support to all lemmas, and we exhibit more sophisticated examples and their translations in Chapter 5.

2.2.5 Grammar

We now give an overview of almost the entirety of the CN specification language. In CN, top-level definitions are either annotated C function definitions, lemma declarations, specification-level function definitions, resource predicates, or datatype definitions.

$\langle toplevel\_def \rangle ::= \langle c\_function\_def \rangle \mid \langle lemma\_def \rangle$
$\quad \mid \langle spec\_predicate\_def \rangle \mid \langle spec\_function\_def \rangle \mid \langle datatype\_def \rangle$

Annotated C function definitions consist of an output type, a list of typed inputs, as

well as `requires` and `ensures` conditions describing the behaviour of the annotated C function in terms of pre- and post-conditions. Conditions may consist of `take`-resource bindings, *constraints*, and let-bindings. Resources are the ownership predicates `Owned`, `Block`, and `each`, corresponding to initialised/uninitialised/iterated points-to predicates indexed by C-types, or user-defined resource predicates. Constraints are boolean-typed expressions that may be universally quantified using the `each` keyword.

⟨*c_function_def*⟩ ::= ⟨*cty f (cty id, .., cty id)*⟩
    `/*@ requires` ⟨*conditions*⟩ `ensures` ⟨*conditions*⟩ `@*/` ⟨*block*⟩

⟨*condition*⟩ ::= `take` ⟨*id*⟩ = ⟨*resource*⟩ | ⟨*constraint*⟩ | `let` ⟨*id*⟩ = ⟨*t*⟩

⟨*conditions*⟩ ::= ⟨*condition*⟩; ...; ⟨*condition*⟩;

⟨*resource*⟩ ::= p(t,...,t) | `each` (⟨*bty*⟩ ⟨*id*⟩; t) `{` p(t,...,t) `}`

⟨*pred_name, p*⟩ ::= `Owned <` ⟨*cty*⟩ `>` | `Block <` ⟨*cty*⟩ `>` | ⟨*id*⟩

⟨*constraint*⟩ ::= t | `each` (⟨*bty*⟩ ⟨*id*⟩; t) | `{` t `}`

Lemmas resemble annotated C function definitions, except they do not have an output type or a C function body.

⟨*lemma_def*⟩ ::= `lemma` ⟨ *f (cty id, .., cty id)*⟩
    `/*@ requires` ⟨*conditions*⟩ `ensures` ⟨*conditions*⟩ `@*/`

Specification-level function definitions and inductive predicate definitions are user-definable and permitted to be mutually recursive. Function definitions consists of an output type, a list of typed inputs, and a function body consisting of a term.

⟨*spec_function_def*⟩ ::= `function` ⟨*bty*⟩ ⟨*id*⟩ (⟨*bty*⟩ ⟨*id*⟩, ..., ⟨*bty*⟩ ⟨*id*⟩) `{` t `}`

⟨*spec_predicate_def*⟩ ::= `predicate` ⟨*bty*⟩ ⟨*id*⟩ (⟨*bty*⟩ ⟨*id*⟩, ..., ⟨*bty*⟩ ⟨*id*⟩) |`{` ⟨*specs*⟩ `}`

⟨*specs*⟩ ::= ⟨*spec*⟩ | `if` (t) `{` ⟨*spec*⟩ `}` `else {` ⟨*specs*⟩ `}`

⟨*spec*⟩ ::= ⟨*conditions*⟩ `return` ⟨*e*⟩;

CN's base types may occur in resources and constraints. They include C arithmetic types such as unbounded integers and fixed-width integers, untyped pointers, compound types such as structs and tuples, as well as user-defined datatypes.

⟨*base_type, bty*⟩ ::= ⟨*id*⟩ | `void` | `bool` | `integer` | `u` nat
    | `i` nat | `pointer` | `struct` tag | `tuple` ⟨*bty, ..., bty*⟩
    | `{` ⟨*bty*⟩ f, ..., ⟨*bty*⟩ f `}` | `list` ⟨*bty*⟩ | `set` ⟨*bty*⟩ | `map` ⟨*bty,bty*⟩ | `datatype` tag

⟨*datatype_def*⟩ ::= `datatype` tag `{` ⟨*constructor*⟩, ..., ⟨*constructor*⟩ `}`

⟨*constructor*⟩ ::= C `{` ⟨*bty*⟩ f, ..., ⟨*bty*⟩ f `}`

CN terms are pure expressions that do not manipulate the heap, except when referencing pointee values in `take`-bound resources. They include unary and binary operators, accesses/updates to structs, records, and tuples, pattern matching, function application, and pointer operations such as `arrayshift` and `membershift`.

⟨*term, t*⟩ ::= k | ⟨*id*⟩ | ⟨*unop*⟩ t | t ⟨*binop*⟩ t | id(t, ..., t)
| `let` ⟨*id*⟩ = t; t' | `if` (t) `{` t `}` `else` `{` t `}` | `match` t `{` `match_cases` `}`
| `each` (⟨*bty*⟩ ⟨*id*⟩ : ⟨*int_range*⟩; t) | (⟨*bty*⟩) t | &⟨*id*⟩ | `array_shift` ⟨*cty*⟩(t,t)
| `member_shift`⟨*tag*⟩(t,f) | `good` ⟨*cty*⟩(t) | `sizeof` ⟨*cty*⟩ | `offsetof`(tag,f) | t[t']
| t[t : t, ..., t : t] | `{` f : t,...,f : t `}` t.⟨*id*⟩ | `{` f : t, ..., f : t..t `}`
| C `{` f : t, ..., f : t `}` | `[` t,...,t `]` | `default` ⟨*bty*⟩

⟨*match_case*⟩ ::= ⟨*pattern*⟩ `=>` `{` t `}`

⟨*pattern*⟩ ::= _| ⟨*id*⟩ | C `{` f : ⟨*pattern*⟩, ..., f : ⟨*pattern*⟩ `}`

⟨*unop*⟩ ::= ! | -

⟨*binop*⟩ ::= `*` | `/` | `+` | `-` | `==` | `!=` | `<` | `>` | `<=` | `>=` | `&&` | || | `==>`

⟨*int_range*⟩ ::= k,k

This concludes our overview of CN's syntax. We note that while CN lemmas only contain C-types, IDs, and conditions, conditions can contain references to top-level definitions such as inductive predicates, specification-level functions, and datatype definitions. This means that in practice, the lemma translation mechanism implemented in this project must supply Rocq translation for all CN language constructs presented above, save for annotated C function definitions. We present our full translation in Section 3.

2.3 The Rocq Proof Assistant

The Rocq proof assistant is the target of our translation. Rocq, previously known as Coq, is an interactive theorem prover based on the Calculus of Constructions [7].

2.3.1 Iris

Iris is a framework for concurrent separation logic implemented in Rocq, through which the separation logic constructs described in Section 1 become available as translation targets. Below are the connectives available in Iris relevant to CN:

⟨*P,Q,R*⟩ ::= `True` | `False` | ⟨*P*⟩ ∧ ⟨*Q*⟩ | ⟨*P*⟩ ∨ ⟨*Q*⟩ | ⟨*P*⟩ ⇒ ⟨*Q*⟩
| ∀ x. ⟨*P*⟩ | ∃ x. ⟨*P*⟩ | ⟨*P*⟩ ⋆ ⟨*Q*⟩ | ⟨*P*⟩ −⋆ ⟨*Q*⟩ | t = u | ...

An Iris *proof mode* exists and extends Rocq to include named proof contexts for separation logic [18].

While the higher-order separation logic implemented in Iris is *generic* over the language in which one reasons about using Hoare triples, we note that CN lemmas are not actually Hoare triples (see Section 2.2.2) and that our translation mechanism does not require reasoning about a particular language. Therefore, we wish to clarify to readers familiar with Iris that our implementation only makes use of the availability of separation logic connectives and their associated proof rules in Iris, which is a fragment of Iris's full capabilities.

Chapter 3

Related work

In this chapter we discuss the literature on separation logic, the theoretical foundations of CN, as well as the Iris separation logic framework for Rocq. We then survey the existing literature on CN as well as similar research efforts towards the verification of imperative languages, with a focus on how these efforts integrate manual proofs carried out in theorem provers like Rocq.

Separation logic is a logic that facilitates reasoning about programs that directly manipulate pointers by providing a mathematical model of computational resources and their *ownership*. It is an extension to Hoare logic [13] first presented by Reynolds [32] based on previous work by Burstall [6], and independently discovered by O'Hearn and Ishtiaq [14] based on existing work in the logic of "bunched implications" by O'Hearn and Pym [29]. The Iris framework implements a higher-order concurrent separation logic in the Rocq theorem prover [17], making machine-checkable separation logic proofs possible. Iris has been utilised in verification efforts such as as the development of a program logic for the verification of *object compatibility patterns* [35], a logical relations model for the verification of concurrent higher-order imperative programs with continuations [37], and *RustBelt*, a safety proof for a realistic subset of the Rust programming language [16].

CN is a verification tool for C code designed with the goal of predictable proof automation, first presented by Pulte et al. in 2023 [30]. The tool is available for public use and is accompanied by official documentation containing a language reference and examples of appropriate usage [31]. The *Fulminate* system [2] is a runtime testing system for CN specifications first presented by Banerjee et al. in 2025.

CN is based on Cerberus, a realistic semantics for ISO C [23] [24] [25]. Cerberus is also the basis of *RefinedC* [33], a type system for C implemented inside Rocq. Like CN, verification in RefinedC is carried out through user-supplied annotations to C code, but all such annotations are directly translated into Rocq, while CN only exports lemmas to Rocq in situations where the SMT backend may be insufficient. In RefinedC, the C program itself is translated into an embedding of C in Rocq called *Caesium*, and automated proof

search/manual proof is used to typecheck the resulting Caesium code against its translated annotations in Rocq. RefinedC also integrates the *VIP* memory model [19], which CN developers are currently experimenting with.

Verifast [15] is another verification tool using separation logic specifications, targeting C and Java. Verifast does not involve Rocq, and verification is based on *forward symbolic execution*, an algorithm that uses symbolic states containing SMT terms for verification. Verifast supports *lemma functions* similar to CN lemmas which are also annotation-level statements that supply pieces of reasoning involving inductive datatypes and proofs because they are not SMT-solvable. However, these lemmas and their proofs live within Verifast's annotation language rather than Rocq.

Frama-C [34] is a platform for code analysis with verification capabilities, targeting C. Frama-C's verification procedure is based on Hoare logic instead of separation logic, and reasoning steps not solvable by its built-in solvers are discharged to Rocq in a similar manner as CN [4]. Out of the research efforts surveyed here, the Rocq code generated by Frama-C is the most similar to the Rocq code generated by the CN-to-Rocq mechanism implemented in this project. However, it is not a particularly useful reference because Frama-C's generated Rocq code do not involve the Iris separation logic framework.

The *Verified Software Toolchain* (VST) [1] is a project dedicated towards the end-to-end verification of C programs that integrates CompCert, a verified compiler for a large subset of C with a proof of correctness in Rocq [20]. Like CN, VST's verification process also uses separation logic. However, it employs an implementation of separation logic in Rocq that predates Iris [9], and prioritises the manual development of separation logic proofs over automation.

Some other verification projects based on Rocq include the verification of the *CertiKOS* kernel [11], the *SeKVM* hypervisor architecture [21], and the BedRock hypervisor [22], out of which the latter is the most relevant to our project because it uses Iris, but little additional information is available because it is proprietary.

Chapter 4

Design and implementation

In this chapter we set out the intended behaviour of the CN lemma translation mechanism, offer overviews of the CN codebase and the existing incomplete implementation of the translation, and present our implementation with full technical details. The source code to our implementation may be found in the directory

`cerberus/tree/master/backend/cn/lib`

in the code accompanying this submission.

4.1 Project Specification

The main objective of our implementation is to extend the CN codebase such that a call to CN such as `cn verify c_file.c --lemmata theories/Gen_Spec.v` verifies a hypothetical piece of C code `c_file.c` against its CN annotations, and generates a Rocq theory file called `Gen_Spec.v` containing all lemmas requiring manual proof.

From this, we may derive a list of specifications the implementation must satisfy:

1. The generated file should support CN-to-Rocq translations for all language constructs that may occur within a CN lemma.
2. The generated file must contain Rocq translations of the *global typing context* of the CN lemma, which comprises of definitions of datatypes, functions, structs, and resource predicates, such that any lemma statement and its proof may refer to them.
3. The generated file must be accompanied by a library of useful Rocq definitions corresponding to C langauge features such as integer wrapping, CN langauge features such as resource ownership predicates, and an instantiation of Iris that contains the memory model we expect users to work with.
4. The generated file must be set up in a manner that explicitly checks a user-supplied

proof against the translated lemma statement.

Beyond concrete features like the above, the implementation should also be designed with modularisation and maintainability in mind such that it may be easily updated with new translations for CN language constructs and memory models by future developers.

4.2 Existing Work

Prior to the work done for the project as will be described in this report, CN was already able to generate Rocq theory files from a CN lemma, but with some caveats that we will address in terms of the specifications given above.

4.2.1 Overview

We begin by discussing the output structure of the existing implementation. Currently, calls to `cn verify` with the appropriate flags generate a file called `Gen_Spec.v`. This generated Rocq file is structured into four Rocq modules and module types:

- `Types` is a module containing translations of all datatype definitions within the global typing context of the lemma. An inductive datatype definition is directly translated as an inductive Rocq `Type` with identical constructors. Mutually recursive datatypes are directly translated into mutually recursive type definitions in Rocq using the `with` keyword.
- `Parameters` is a module type containing type signatures for all CN *uninterpreted functions* in the global typing context. A CN uninterpreted function is a function that has no body, and it is translated as a Rocq `Parameter` followed by the type signature containing its argument types and return type. Users are expected to supply definitions for these functions when they prove translated CN lemmas in Rocq. This is enforced by the fact that all following modules are paramaterised over `Parameters`, and any instantiation of the `Lemma_Spec` module that does not supply such function definitions will fail to typecheck in Rocq.
- `Defs`, a module parameterised over the `Parameters` module type. This module contains all function definitions available in the global typing context, as well as the statement of the lemma itself, which is translated as a Rocq `Prop` with a type corresponding to the CN type of the lemma.
- `Lemma_Spec` is the module type that the user is expected to instantiate when they prove the lemma. It is parameterised over `Parameters`, imports all definitions from `Defs`, and contains a single Rocq `Parameter` with the type of the lemma statement as defined in `Defs`.

The code for generating the Rocq file described above resides in the file `lemmata.ml`,

within the CN codebase, which is written in OCaml. Its primary philosophy is to recursively pattern-match on the CN language expressions it encounters when fed a lemma, and generates Rocq syntax directly from it. The translation is carried out lazily using a state monad because the CN codebase did not supply the dependency order of CN datatypes and functions back when it was implemented.

4.2.2 Limitations

Now, we discuss the limitations of the existing codebase in terms of the project specifications introduced in section 4.1. The main problem is that the existing implementation is incomplete because while it is able to translate *pure* lemmas, it does not have support for resource predicates or lemma statements concerning resource ownership, which are vital for reasoning about memory management in C. Therefore, the existing implementation does not achieve our first two specifications: that it should have support for all CN language constructs, and that it should generate Rocq translations of the entire global typing context of a given lemma.

However, the existing implementation does offer this project an excellent starting point by offering Rocq translations for most pure CN terms, a small library `CN_Lib.v` containing a lemma about the wrap-around behaviour of C integers, and the module structure for the output file. In so, it partially satisfies the third and fourth specifications concerning the implementation of a library and supplying a module structure for the generated syntax.

Finally, the existing implementation's translation strategy of directly mapping CN terms onto Rocq syntax is unviable because a particular CN term can have several different Rocq translations depending on its context. For example, `IndexTerms.Not` is the CN term for the unary negation operator. This term appears in two different scenarios:

1. Function definitions and if-conditions, where it should have type `Coq.Bool` and therefore be translated as as the string `negb`, and

2. Lemma statements, where it should have type `Prop` and therefore be translated as $\sim$, the tilde unicode character.

The existing implementation distinguishes between the two scenarios above using a boolean flag that is switched on when the main translation function `it_to_coq` is called from the context of a function definition or an if-condition. Extending the same approach towards translations of CN terms not yet implemented in this codebase will lead to the addition of more flags, which will make the code even more difficult to read than it already is. Ideally, some balance is found between the effectiveness and maintainability.

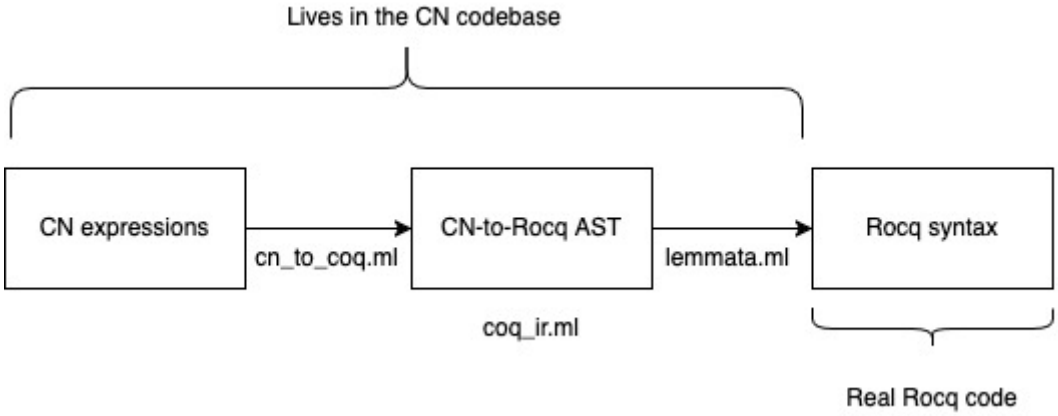
4.3 Implementation

Now, we present an overview of our new implementation of the CN-to-Rocq lemma translation mechanism, and discuss its key differences with the existing implementation. Our implementation resides in three files, `cn_to_coq.ml`, `lemmata.ml`, and `coq_ir.ml`. The design principle here is to divide the translation process into two stages:

1. `cn_to_coq.ml` handles the translation of CN terms into a simple abstract syntax tree (AST) designed for the subset of Rocq used in our translation.

2. `lemmata.ml` handles the translation of our AST into Rocq syntax. This entails directly printing the Rocq syntax corresponding to each AST node in the correct order, surrounded by syntax for module definitions inherited from the old implementation.

The following diagram illustrates the translation process we implemented.



We now discuss our source language of CN expressions in more detail. Importantly, these CN expressions are the existing implementation of the language in OCaml, not the concrete syntax of the CN specification language as described in Section 2. Therefore, we do not need to re-implement a parser for the language. Additionally, due to the part of the pipeline at which lemma generation occurs in the CN codebase, we have the luxury of assuming that all CN expressions we need to translate are well-typed and do not need to be re-validated.

CN expressions are provided to our implementation through the CN global typing context, which contains all definitions that should be available to the user when they prove a translated lemma in Rocq. The global typing context is implemented in the CN codebase as an OCaml record with the following fields, which contains fully-formed CN expressions and the dependency order of constructs like datatypes and functions.

$$
\begin{aligned}
global.t : \{ & struct\_decls \\
& datatypes \\
& datatype\_constrs \\
& datatype\_order \\
& resource\_predicates \\
& resource\_predicate\_order \\
& logical\_functions \\
& logical\_function\_order \\
& lemmata\}
\end{aligned}
$$

We implement an intermediate representation, the CN-to-Rocq AST, in the file `coq_ir.ml`. Its purpose is to split the translation task into two manageable chunks: extracting the data necessary for the translation from existing CN expressions, and printing concrete Rocq syntax. This means the AST is designed with the goal of re-organising information contained in CN terms to mirror the output format of our generated Rocq file so that the reasoning work carried out in `lemmata.ml` is minimised. For example, the top-level of the AST reflects the module structure of the Rocq file we would like to generate, while also corresponding to the four CN top-level language constructs relevant to our translation:

$\langle global \rangle ::= \langle datatype \rangle \mid \langle function \rangle \mid \langle resource\_predicate \rangle \mid \langle lemma \rangle$

Finally, our target language of Rocq syntax is a mixture of pure Rocq as well as Iris syntax. This is because as described in Section 2, the separation logic constructs underlying CN resource predicates are only available through Iris, which is why we take the approach of only translating CN language features that cannot contain resources, such as datatype and function definitions, into pure Rocq. Everything else, including resource predicates and lemma statements, is moved into an Iris *spatial context* and mapped to slightly different syntax: we now surround pure Rocq expressions with special brackets to embed them into Iris, and the separating conjunction and magic wand are both available for use.

In the following sections we present the entirety of our CN-to-Rocq AST with examples of the Rocq syntax we generate from it.

4.3.1 Datatypes

The global typing context fields *datatypes*, *datatype_constrs*, and *datatype_order* respectively contain the name and arguments of all CN datatype definitions within the current environment, the argument types of all CN datatype constructors, and their dependency order. The latter is given as a list of lists where each element represents a group of mutually recursive datatypes.

In our AST, datatypes declarations consist of a name, a list of constructors, and a list of base types for the constructor arguments:

⟨*datatype*⟩ ::= `Datatype` ⟨*name*⟩ ⟨*constructor*⟩*

⟨*constructor*⟩ ::= `DatatypeConstr` ⟨*name*⟩ ⟨*base_type*⟩*

Here, the OCaml-level constructors `Datatype` and `DatatypeConstr` are used to indicate the concrete Rocq syntax that the term should be translated into. In the context of datatype declarations, this entails printing the Rocq keyword `Inductive` before the datatype's name, adding bars before constructor names, and so on. This part of our translation is very straightforward. To illustrate, below is a simple CN specification datatype declaration for `List`:

```
/*@ datatype List {
  Nil {},
  Cons {i32 Head, datatype List Tail}
}
@*/
```

And the Rocq translation we generate for this definition is:

```
Inductive
    List : Type :=
    | Nil : List
    | Cons : Z -> List -> List.
```

We note that we translate bitvectors as integers, a decision inherited from the previous implementation. We discuss the implications of this decision in section 4.3.3. As with the original implementation, we also print all datatype declarations in a Rocq module `Types` at the top of the generated Rocq file.

4.3.2 Functions

The *logical_functions* and *logical_function_order* fields in the CN global typing context respectively contain all functions within the current environment, and the dependency order of mutually recursive groups of functions.

CN supports three types of functions that we must treat differently in our translation: uninterpreted functions, defined functions, and recursively defined functions. Function definitions are then divided into defined and uninterpreted functions (representing C's external functions). This distinction is necessary because defined functions should be translated with different Rocq syntax depending on whether they are recursive. In both cases, the fields in their AST constructors consist of a name, a list of named arguments, and a return type. Then, interpreted functions have a function body consisting of a term, while uninterpreted ones do not. The relevant part of the AST is given below.

⟨*function*⟩ ::=

`InterpFun` ⟨*name*⟩ (⟨*name*⟩,⟨*base_type*⟩)* ⟨*base_type*⟩ ⟨*uninterpreted_fun*⟩

| `UninterpFun` ⟨*name*⟩ (⟨*name*⟩,⟨*base_type*⟩)* ⟨*base_type*⟩ ⟨*interpreted_fun*⟩

⟨*interpreted_fun*⟩ := `Def` ⟨*term*⟩ | `RecDef` ⟨*term*⟩

⟨*uninterpreted_fun*⟩ := `Uninterp` | `Uninterp_prop`

Our AST makes the distinction between `Uninterp` and `Uninterp_prop`, respectively representing computational and propositional uninterpreted functions. This distinction is necessary because Rocq uses a different syntax for functions on the propositional level, and our `Uninterp_prop` AST node flags this for the syntax generation stage.

As discussed in section 4.2.1, uninterpreted functions are translated as Rocq parameters in a module type `Parameters` dedicated to them, and all subsequent modules are parameterised over it. Defined functions and recursively defined functions are translated the same way, except we append different Rocq syntax before them: `Definition` for the former and `Fixpoint` for the latter. Similarly, groups of mutually recursive functions are interspersed by the `with` keyword. These definitions are printed to the `Defs` module.

4.3.3 Resource Predicates

To translate a CN resource predicate definition into Rocq, we first describe `CN_Lib_Iris`, a library of Iris-related definitions we implement. `CN_Lib_Iris` contains the following:

- The Iris *resource algebra* that our instantiation of Iris is parameterised over,
- Rocq definitions corresponding to CN's `Owned`, `Block`, and `Each` constructs, and
- Rocq definitions corresponding to CN built-in functions such as `arrayshift`.

A file containing this library is always exported alongside the Rocq file containing our translations, such that our translated resource predicates and lemmas can refer to it. We address each of the features listed above before moving on to discuss our translations for resource predicates.

Resource Algebras

We omit the formal definition of resource algebras in the interest of space. Informally, resource algebras make resource ownership predicates available in Iris, and they enforce properties such as the associativity and commutativity of resource ownership composition to ensure the points-to predicate interacts well with the separating conjunction [17].

An Iris spatial context is parametrised over Σ, a set of appropriate resource algebras selected by the user. This parametrisation is necessary because resource algebras can contain types of resource ownership that go beyond the points-to predicate introduced in Section 1 of this report. For example, a resource algebra may implement a version of

the points-to predicate with *fractional permissions*, written $t_1 \overset{z}{\mapsto} t_2$ where t_1 is a heap location, t_2 is a value, and z is a rational number satisfying $0 < z \leq 1$, such that $z = 1$ represents total ownership, while $0 < z < 1$ represents partial ownership such as read-only access to t_1 [5]. For the purposes of our implementation, we instantiate Iris with a resource algebra called `gen_heapGS` which is shipped alongside Iris as part of the example language *HeapLang* [8]. This resource algebra implements the fractional permission model, through which we can recover the simpler points-to predicate described in Section 1 by defining a Rocq notation that sets z to 1 at all times, like so:

```
Notation "l ↦ v" := (pointsto l (DfracOwn 1) v) (at level 20) : bi_scope.
```

We stick to this definition of the points-to predicate throughout our implementation.

Ownership Predicates

We recall from Section 2.2.2 that `take P = Owned<t>(p)` corresponds to a points-to predicate `p` $\mapsto_{\texttt{t}}$ `P` for some pointer `p`, value `P`, and C-type `t`, where `P` is universally quantified if the predicate appears in a `requires` clause, and existentially quantified if it appears in a `ensures` clause. The type parameter `t` can be almost any C-type, and `P` is a value of that type.

Our translation represents pointers as Rocq signed integers `Z` and values as `option Z`, where the `None` constructor represents an owned but uninitialised memory cell. The null pointer is represented as the integer 0 and cannot appear on the left-hand-side of a points-to predicate. The resulting definition are the following, where we first implement `Owned` and `Block` predicates for individual heap cells. We implement `Block l` to be true whenever `Owned l v` for some `(l : Ptr)` and `(v : Z)` to reflect the fact that in CN, `Owned` is a subtype of `Block`.

```
1    Definition Owned (l: Ptr) (v : Z) : iProp Σ := (l ↦ Some v) ∧ ⌜l ≠ 0⌝.
2    Definition Block (l :Ptr) : iProp Σ := ((l ↦ None) ∧ ⌜l ≠ 0⌝) ∨
3                                           (∃ v,l ↦ Some v).
```

Neither of the predicates defined above are directly used in our translation. Instead, we use them as components for building points-to predicates indexed by a C primitive type `t`, which is expressed as ownership of n consecutive memory cells, where n is the size of `t`. For example, ownership predicates for values of C-type `char` are defined like the following, where each heap cell corresponds to one byte of memory:

```
1  Definition Owned_char (p: Ptr) (v : Z) : iProp Σ := Owned p v.
```

Similarly, an ownership predicate for a signed integer `int` occupy four bytes of memory, which we represent as the separating conjunction over four consecutive heap cells. Because our implementation currently lacks support for bitvectors, we express the values stored

in each heap cell through the following function:

```
Definition int_to_bytes (val : Z) : (Z * Z * Z * Z).
    Admitted.
```

We leave `int_to_bytes` undefined in anticipation of better bitvector support, which we discuss at the end of Chapter 5. We then define ownership of an integer value as:

```
Definition Owned_int (l: Ptr) (v : Z) : iProp Σ :=
  let '(b1, b2, b3, b4) := (int_to_bytes v) in
  Owned l b1 ⋆
  Owned (l + 1) b2 ⋆
  Owned (l + 2) b3 ⋆
  Owned (l + 3) b4.
```

Iterated Ownership Predicates

Next, we recall from Section 2.2.2 that predicates of the form `each(r){A}` correspond to the separating conjunction over every instance of the assertion `A` for which the guard `r` holds. While we don't have a general means of translating all `each` predicates, we observe that `each` predicates are usually used to reason about ownership of arrays, where they take the following form:

```
each(i32 j; min <= j && j < max)
        { Owned<int>(array_shift<int>(p,j)) };
```

Interpreting `p` as the base pointer of an `int` array, this predicate takes ownership of the array cells `j` between `min` and `max`. In our translation, we employ a library function `each_int` which takes as input the lower bound `min`, the length of the desired array segment `len`, a base pointer `p`, and a list of integers `l` containing exactly the values stored in the array. Its definition is the following.

```
Fixpoint each_int (min len : nat) (p : Ptr) (l : list Z) : iProp Σ :=
  (match len with
    | O => emp%I
    | S n => (match l with
      | cons x xs => (Owned_int (arrayshift p 4 min) x ⋆
                          each (S min) n p xs)%I
      | nil => False%I
      end)
  end).
```

This function generates `Owned_int` predicates for each element in an array segment of length `len` with base pointer `array_shift<int>(p,j)` (where setting `j` to 0 makes the base pointer `p`). The elements of the array themselves are represented through a list `l`, implemented with Rocq's built-in `list` type rather than the `List` inductive type we

generated previously from CN definitions. The translation of arrays to lists is motivated by the fact that we do not have translation support for maps. The body of the function states that if `len` is not zero, we check that `l` is not empty, assert ownership of its first element at `array_shift<int>(p,j)`, and take its separating conjunction with a recursive call that increments `j`, the number of positions we shift `p` by. This results in a separating conjunction over `len` owned integers stored between the pointers `array_shift<int>(p,j)` and `array_shift<int>(p,j + n)`, and that the list `l` contains exactly these integers in the correct order.

To see this definition used in practice, the following CN expression:

```
take A = each(i32 j; min <= j && j < max)
        { Owned<int>(array_shift<int>(p,j)) };
```

is translated as the following Rocq predicate if appearing in a `requires` clause (it is quantified existentially if appearing in a `ensures` clause or resource predicate):

```
∀ (A : list Z), each_int min (max - min) p A
```

We emphasise that `each_int` does not take `max` as an argument, but rather `(max - min)`.

Ownership of Structs

While ownership predicates for values of primitive C types can be supplied as library functions, we also need to generate ownership predicates for C structs on-demand because they need to be defined for arbitrary user-defined structs. Therefore, we divert our attention from the contents `CN_Lib_Iris` briefly to discuss how structs and their ownership predicates are handled.

The *struct_decls* field of the CN global typing context contains names and definitions of all C structs relevant to a lemma. Because structs and struct ownership predicates can be referred to by name in function arguments and lemma statements, our Rocq translation prints definitions for all structs and their respective ownership predicates in the `Types` module, preceding all definitions that may call them.

Structs themselves are translated as Rocq records with identical fields. For example, a C struct `point` with the following definition

```
1    struct point { char x; int y; };
```

Is translated as the following Rocq record:

```
1    Record point : Type := { x : Z; y : Z; }.
```

Struct field accesses can then be directly translated as the Rocq record field accesses `point.(x)` and `point.(y)`. This is in contrast to the existing implementation's approach

where structs are translated as appropriately-typed tuples, such that a struct's fields can be accessed using tuple projections. While this strategy also works, it renders field names anonymous and makes the translated struct's correspondence with the original C definition of that struct less obvious.

Ownership predicates for structs are generated right after their definitions. In C, the size of a struct is the sum of the size of it fields, possibly interspersed by padding bytes for data alignment. Therefore, our translation of a struct ownership predicate is the separating conjunction over the ownership predicates for each of the struct's fields and padding bytes. For example, the ownership predicate we generate for `point` is the following:

```
Definition Owned_point (l: Ptr) (v : point) : iProp Σ :=
      Owned_char (CN_Lib_Iris.shift l 0) v.(x)
    ⋆ padding (arrayshift l 1 1) 3
    ⋆ Owned_int (CN_Lib_Iris.shift l 4) v.(y).0
```

We discuss the new library functions used in this definition below.

Additional Library Functions and Lemmas

The following library functions are defined in `CN_Lib_Iris`:

- `CN_Lib_Iris.shift p n` shifts a pointer `p` by `n` bytes, implemented as their sum.
- `padding p n` denotes ownership of `n` empty bytes at location `p`, implemented as the separating conjunction over `n` consecutive `Block` predicates starting at `p` and ending at `p + n`.
- `arrayshift p m n`, corresponds to array-shifts in C where a pointer `p` is shifted by `n` bytes for `m` times.

Information about the size of each type, as well as the pointer arithmetic involved in determining the number of padding bytes needed in a struct definition, are all supplied by our source CN expressions and do not need to be re-validated.

In addition, `CN_Lib_Iris` contains a number of lemmas about the behaviour of ownership predicates. These include properties such as the fact that an owned pointer cannot be null, that two separately owned pointers cannot be equal, and that two ownership predicates for the same pointer must agree on its pointee value. The proofs for several of these lemmas rely on proofs supplied by the library of lemmas accompanying the resource algebra we used to make the points-to predicate available in our implementation, and they become useful in the case studies discussed in Chapter 5 of this report.

Resource Predicates

Now, we return to discussing how our implementation translates resource predicates. Our AST for resource predicates is the following:

⟨*resource_predicate*⟩ ::=
`InterpPred` ⟨*name*⟩ ⟨*name*⟩ (⟨*name*⟩,⟨*base_type*⟩)* ⟨*base_type*⟩ ⟨*clause*⟩*
| `UninterpPred` ⟨*name*⟩ ⟨*name*⟩ (⟨*name*⟩,⟨*base_type*⟩)* ⟨*base_type*⟩

⟨*clause*⟩ ::= `Clause` ⟨*term*⟩* ⟨*term*⟩

We have different constructors for interpreted and uninterpreted resource predicates, where uninterpreted resource predicates are resource predicates with no body. In both cases, the OCaml-level constructor fields are respectively the name of the predicate, the name of the input pointer to the predicate, a list of additional input names and base-types, and the output type of the predicate. Interpreted predicates then have a body consisting of a list of clauses, each representing a guarded case in an if-statement. Guards are explicitly given in a list before the body of each clause because our translation has to generate the negation of guards in all previous clauses of the if-statement to ensure each case is mutually exclusive.

We now display a CN inductive resource predicate and its translation. The CN predicate `BothOwned`, defined below, takes two pointers `p` and `q` as input. If the two pointers are equal, the predicate asserts ownership of `p`. If they are not equal, ownership is asserted for both pointers. Then, the predicate returns an anonymous record comprised of their pointee values.

```
1  /*@
2  predicate { u32 P, u32 Q } BothOwned (pointer p, pointer q){
3    if (ptr_eq(p,q)) {
4      take PX = Owned<unsigned int>(p);
5      return {P: PX, Q: PX};
6    }
7    else {
8      take PX = Owned<unsigned int>(p);
9      take QX = Owned<unsigned int>(q);
10     return {P: PX, Q: QX};
11   }
12 }
13 @*/
```

The translation we generate for `BothOwned` is below. We recall from Chapter 2 that all ownership predicates appearing in inductive resource predicates are existentially quantified [32]. We take the logical conjunction of a guard and its associated clause body, and default to using separating conjunctions within clause bodies. Pure assertions are embedded into Iris proof contexts using brackets ⌜`like so`⌝.

```
Fixpoint BothOwned (p : Ptr) (q : Ptr) (ret : (Z * Z)) : iProp Σ :=
  (⌜ (p = q) ⌝ ∧
    ∃ (PX : Z), Owned_int p PX ⋆ ⌜ret = (PX, PX)⌝) ∨
  (⌜ (Is_true true) ⌝ ∧ ⌜ ¬(p = q) ⌝ ∧
    ∃ (PX : Z), Owned_int p PX ⋆
    ∃ (QX : Z), Owned_int q QX ⋆ ⌜ret = (PX, QX)⌝).
```

In predicates like `BothOwned`, we translate if-statements as a disjunction of conjunctions, where each conjunct consists of a clause's guards and body. We also note that anonymous records do not have a direct analogue in Rocq, so we translate them as tuples.

An important caveat to our translation of inductive resource predicates is that they do not immediately pass Rocq's termination checker, which is because the recursive calls in these predicates do not always occur on a syntactically smaller term. Therefore, Rocq does not accept the resource predicates we generate unless we disable its termination checker using the command `Unset Guard Checking`. While this is unsafe, we are careful to only disable termination checking on resource predicate definitions, which we believe are terminating. To justify this, we supply an informal sketch below for a proof that our inductive resource predicates must have a least fixed point.

Proof sketch

Definition 4.3.1 (Complete lattice)**.** A **complete lattice** $(L, \leq)$ is a partially ordered set for which each subset of L has a greatest lower bound (meet) and least upper bound (join). As a consequence, complete lattices cannot be empty.

Definition 4.3.2 (Monotonic function)**.** A **monotonic function** f over a complete lattice $(L, \leq)$ is a function satisfying $\forall\, x, y \in L,\ x \leq y \implies f(x) \leq f(y)$.

Theorem 1 (Knaster-Tarski theorem)**.** *For a complete lattice $(L, \leq)$ and monotonic function $f : L \to L$, the set of fixed points of f in L is also a complete lattice.*

Corollary 1.1. Any monotonic function over a complete lattice has a least fixed point.

Theorem 2. *Rocq translations of CN inductive resource predicates have a least fixed point.*

Proof sketch. We show that our Rocq translation of CN inductive resource predicates have least fixed points by first showing that monotone functions between translated CN resource predicates have least fixed points, then showing that translated inductive resource predicates are instances of such monotone functions.

CN resource predicates ($\tau \to \sigma$ **pred**) are predicates over an output of a resource of type σ, given an input of type τ (see Section 2.2.2 on the input-output distinction in CN). In our translation from CN to Rocq, we interpret CN resource predicates as

$$\tau \to \sigma \to \texttt{iProp}\ \Sigma$$

where `iProp` Σ are Iris predicates parametrised over a set of resource algebras Σ.

If $(L, \leq)$ is a complete lattice and X is a set, then $X \to L$ is also a complete lattice pointwise. Therefore, $(\tau \to \sigma \to \texttt{iProp})$ is a complete lattice because $\texttt{iProp}\ \Sigma$ is a complete lattice [3]. Then, monotone functions

$$(\tau \to \sigma \to \texttt{iProp}) \to (\tau \to \sigma \to \texttt{iProp})$$

have least fixed points by Corollary 1.1.

We now show that our translated inductive resource predicates are such monotone functions. We do so by considering a CN inductive resource predicate P with input x of type τ as

$$P(x : \tau) = \mathit{body}$$

where *body* is the body of P's definition, containing a recursive call to P.

The syntax of *body* is the following:

$\langle C \rangle ::= \exists x : \tau.\ \langle C \rangle \mid \phi \wedge \langle C \rangle \mid \langle C \rangle \star \langle C \rangle \mid \mathit{emp} \mid P(x)$

The translation of this syntax to Rocq is the obvious one, and we denote the Rocq translation of *body* as $[\![\mathit{body}]\!]$.

We now consider the following Rocq function *Pbody* defined as

$$\mathit{Pbody}\ Q\ x = [\![\mathit{body}]\!]\ [P/Q]$$

where $[P/Q]$ denotes the substitution of the recursive call to P with the Rocq-level variable Q. We observe that *Pbody* has type

$$\mathit{Pbody} : (\tau \to \sigma \to \texttt{iProp}\ \Sigma) \to \tau \to \sigma \to \texttt{iProp}\ \Sigma$$

As the grammar of *body* does not contain negations, implications, or magic wands, *body* always gives rise to monotonic functions on sets of heaps, and the powerset of heaps is a complete lattice. Therefore, *Pbody* has a least fixed point.

We define the semantic translation of CN predicate P as the least fixed point of its translated body:

$$[\![P]\!] = lfp\ (\mathit{Pbody})$$

where lfp denotes the function mapping the semantics of *Pbody* to its least fixed point. Then, we have

$$[\![P]\!] = [\![\mathit{body}]\!][P/[\![P]\!]]$$

Therefore, P is well-defined.

□

Completing the full proof in Rocq and supplying it to Rocq's termination checker should convince Rocq to accept our inductive resource predicates. We do not implement the proof as part of this project due to time constraints.

4.3.4 Lemmas

Finally, we discuss how we translate separation logic lemmas from CN to Rocq. We recall from Section 2 that CN lemmas comprise of `requires` and `ensures` clauses, informally corresponding to user-specified pre- and post-conditions for the annotated function. As CN lemmas do not actually make assertions about the behaviour of a C program, a typical lemma comprised of some `requires` assertions A, B, and `ensures` assertions C, D:

$$\texttt{requires}\ A;\ B;\ \ \texttt{ensures}\ C;\ D;$$

may be more accurately translated into separation logic as the following predicate:

$$A \mathrel{-\!\!\star} B \mathrel{-\!\!\star} C \star D$$

This predicate states that given two separate extensions to the heap for which A and B are respectively true, the separating conjunction of C and D should hold in the resulting heap. More generally, assertions appearing in `requires` clauses are interspersed by magic wands, while assertions appearing in `ensures` clauses are in separating conjunction.

To demonstrate how this works in our implementation, we consider the following lemma, which states that the result of appending the empty list `Nil{}` to a `List L1` should be equal to `L1`.

```
/*@
lemma Append_Nil_RList (datatype List L1)
  requires true;
  ensures Append(L1, Nil{}) == L1;
@*/
```

A lemma is needed in this situation because a proof of the corresponding separation logic predicate, given below, requires a proof by induction on `L1`, which is beyond the capabilities of CN's SMT backend.

$$\forall\ (L1 : List).\ True \mathrel{-\!\!\star} Append(L1, Nil) = L1$$

The translation we generate for this lemma is below, where `Is_true true` embeds the Rocq boolean value `true` into the propositional level at which the lemma statement occurs.

```
Definition Append_Nil_RList_type : iProp Σ :=
    ∀ (L1 : List),
```

```
3     ⌜ (Is_true true) ⌝ -∗
4     ⌜ ((Append L1 (Nil)) = L1) ⌝ ∧ ⌜ Is_true true ⌝.
```

We discuss how lemmas like this can be proven in Chapter 5 of this report. In our AST, lemmas simply consist of their names and the term describing their definitions.

⟨*lemma*⟩ ::= `Lemma` ⟨*name*⟩ ⟨*term*⟩

4.3.5 Terms

Finally, we discuss the rest of our AST, which comprises of the terms that may appear within each of the constructs introduced above.

Our AST takes a "flat" approach to terms, in the sense that apart from terms with obvious classes of subterms such as unary operators and binary operators, we do not specify subterms in a way that exactly matches the syntax of the CN specification language as given in Section 2. For example, we do not enforce that the conditional in an if-statement must be a boolean proposition. This is because the CN expressions supplied to our translation mechanism are already well-formed and well-typed, so any work towards re-validating them at our stage of the pipeline is redundant. Given this design choice, there is nothing particularly interesting about this section of the AST except for the `iris_term` constructor, which is used to indicate that all pure subterms of that term should be surrounded by Iris's pure embedding brackets.

We exhibit the rest of the AST below.

⟨*term*⟩ ::= `Const` ⟨*const*⟩
| `Unop` ⟨*unop*⟩ ⟨*term*⟩ ⟨*base_type*⟩
| `Binop` ⟨*binop*⟩ ⟨*term*⟩ ⟨*term*⟩ ⟨*base_type*⟩
| `Match` ⟨*term*⟩ (⟨*pattern*⟩, ⟨*term*⟩)*
| `IfThenElse` ⟨*term*⟩ ⟨*term*⟩ ⟨*term*⟩
| `eachI` int ⟨*name*⟩ ⟨*basetype*⟩ int ⟨*term*⟩
| `MapSet` ⟨*term*⟩ ⟨*term*⟩ ⟨*term*⟩
| `MapGet` ⟨*term*⟩ ⟨*term*⟩
| `RecordMember` ⟨*term*⟩ ⟨*name*⟩ int int
| `RecordUpd` ⟨*term*⟩ ⟨*name*⟩ ⟨*term*⟩ int int
| `Record` ⟨*term*⟩*
| `StructMember` ⟨*term*⟩ ⟨*name*⟩ int int
| `StructUpd` ⟨*term*⟩ ⟨*name*⟩ ⟨*term*⟩ int int
| `Cast` ⟨*term*⟩ ⟨*base_type*⟩
| `Apply` ⟨*name*⟩ ⟨*term*⟩*
| `Apply_prop` ⟨*name*⟩ ⟨*term*⟩*
| `Good` ⟨*term*⟩
| `Representable` ⟨*name*⟩ ⟨*base_type*⟩ ⟨*term*⟩

| `Constructor` ⟨*name*⟩ ⟨*term*⟩*
| `NthList` ⟨*term*⟩ ⟨*term*⟩ ⟨*term*⟩
| `ArrayToList` ⟨*term*⟩⟨*term*⟩⟨*term*⟩
| `Let` ⟨*name*⟩ ⟨*term*⟩⟨*term*⟩
| `WrapI` int int ⟨*term*⟩
| `ArrayShift` ⟨*term*⟩ int ⟨*term*⟩
| `Forall` ⟨*name*⟩ ⟨*base_type*⟩ ⟨*term*⟩
| `Define` ⟨*name*⟩ ⟨*term*⟩ ⟨*term*⟩
| `Constraint_LRT` ⟨*term*⟩ ⟨*term*⟩
| `Constraint_LAT` ⟨*term*⟩ ⟨*term*⟩
| `I_LAT` ⟨*term*⟩
| `I_LRT`
| `Owned_LRT` ⟨*name*⟩ ⟨*base_type*⟩ ⟨*iris_term*⟩ ⟨*term*⟩ ⟨*term*⟩*
| `Block_LAT` ⟨*name*⟩ ⟨*base_type*⟩ ⟨*iris_term*⟩ ⟨*term*⟩
| `Owned_LAT` ⟨*name*⟩ ⟨*base_type*⟩ ⟨*iris_term*⟩ ⟨*term*⟩ ⟨*term*⟩*
| `Block_LAT` ⟨*name*⟩ ⟨*base_type*⟩ ⟨*iris_term*⟩ ⟨*term*⟩
| `PredName_LAT` ⟨*name*⟩ ⟨*name*⟩ ⟨*base_type*⟩ ⟨*term*⟩*
| `PredName_LRT` ⟨*name*⟩ ⟨*name*⟩ ⟨*base_type*⟩ ⟨*term*⟩*
| `Pure` ⟨*term*⟩

⟨*iris_term*⟩ ::= `IrisTerm` ⟨*term*⟩

⟨*unop*⟩ ::= `Neg` | `NegProp` | `BW_FFS_NoSMT` | `BW_CTZ_NoSMT`

⟨*binop*⟩ ::= `Add` | `Sub` | `Mul` | `Div` | `Mod` | `Rem` | `Lt` | `LtProp` | `Le` | `LeProp` | `Exp` | `BwXor` |
`BwAnd` | `BwOr` | `Eq` | `EqProp` | `And` | `AndProp` | `Or` | `OrProp` | `Impl` | `ImplProp` |

⟨*const*⟩ ::= `Bool` bool | `BoolProp` bool | `Z` z | `Bits` z |

⟨*sign*⟩ ::= `signed` | `unsigned`

⟨*pattern*⟩ ::= `PatName` ⟨*name*⟩ | `PatWildCard` | `PatConstructor` ⟨*name*⟩ ⟨*pattern*⟩*

⟨*name*⟩ ::= `Name` string

⟨*base_type*⟩ ::= `Bool` | `Integer` | `Bits`⟨*sign*⟩ int | `Map` ⟨*base_type*⟩ ⟨*base_type*⟩
| `Struct` ⟨*name*⟩ ⟨*base_type*⟩* | `Loc` | `Datatype` ⟨*name*⟩
| `List` ⟨*base_type*⟩ | `Unit` | `Membyte` | `Real` | `Alloc_id` | `CType`
| `Tuple` ⟨*base_type*⟩* | `Set` ⟨*base_type*⟩

In the AST above, `LRT` and `LAT` suffixes respectively stand for "Logical Argument Type" and "Logical Return Type", which respectively correspond to `requires` and `ensures` clauses. We attach these suffixes to constructors for all ownership predicates to indicate whether they should be printed alongside a universal or existential quantifier. We also note that we currently lack support for the CN base types `Membyte`, `Real`, `Alloc_id`, and `Ctype`.

We omit the full mapping between our AST and Rocq syntax, and direct interested readers

to the file `lemmata.ml` in the codebase supplied alongside this report, which prints the Rocq file containing our translation from an AST supplied by `cn_to_coq.ml`, as well as library imports and module names.

Chapter 5

Evaluation

In this chapter we demonstrate the capabilities of our implementation by using it to prove some lemmas that CN users have historically needed, sourced from the CN User Tutorial [31]. In particular, we consider three case studies: list reversals, equivalence of array ownerships, and imperative queues. Each case study is selected to illustrate a new feature unavailable in the previous implementation. The files associated with each case study can be found in the following directory in the code accompanying this report.

`cerberus/backend/cn/lib/lemma_tests/case_studies`

Following this, we evaluate our CN-to-Rocq translation mechanism by comparing it to the existing implementation through the criteria set out in Chapter 4.

5.1 Case Study 1: List Reversals

For our first example, we consider two pure lemmas concerning list reversals. Assuming the machinery for CN specification-level lists introduced in Chapter 2, we present the Rocq translation of the CN-level function `Append` generated by our implementation below:

```
1  Fixpoint Append (L1 : List) (L2 : List) :=
2    (match L1 with
3      | (Nil) => L2
4      | (Cons H T) => (Cons H (Append T L2)) end).
```

We note that the previous implementation of the translation mechanism was not able to translate `Append` because it lacked support for recursive functions.

Given this definition, we now consider two lemmas `Append_Nil_RList` and `Append_Cons_RList`, the former of which was discussed in Chapter 4, and the latter of which states that appending a `List` in the `Cons` constructor case to another `List L1` is equivalent to `Snoc`-ing

the head of that list to `L1`, then appending its tail to `L1`. We omit the definition of `Snoc`, which adds an element to the end of a `List`.

```
/*@
lemma Append_Nil_RList (datatype List L1)
  requires true;
  ensures Append(L1, Nil{}) == L1;

lemma Append_Cons_RList (datatype List L1, i32 X, datatype List L2)
  requires true;
  ensures Append(L1, Cons {Head: X, Tail: L2})
          == Append(Snoc(L1, X), L2);
@*/
```

The Rocq translations our implementation automatically generates for these lemmas are the following.

```
Definition Append_Nil_RList_type : iProp Σ :=
    ∀ (L1 : List),
    ⌜ (Is_true true) ⌝ -∗ ⌜ ((Append L1 (Nil)) = L1) ⌝ ∧
    ⌜ Is_true true ⌝.

Definition Append_Cons_RList_type : iProp Σ :=
    ∀ (L1 : List), ∀ (X : Z), ∀ (L2 : List),
    ⌜ (Is_true true) ⌝ -∗
    ⌜ ((Append L1 (Cons X L2)) = (Append (Snoc L1 X) L2)) ⌝ ∧
    ⌜ Is_true true ⌝.
```

To supply a proof for these lemmas, the user must instantiate the `Lemma_Spec` module generated by our translation. We supply a partial version of the `Lemma_Spec` module generated by our translation below.

```
Module Type Lemma_Spec (P : Parameters).
  ...

  Parameter Append_Nil_RList : ⊢ Append_Nil_RList_type.

  Parameter Append_Cons_RList : ⊢ Append_Cons_RList_type.

End Lemma_Spec.
```

An instantiation of `Lemma_Spec` can then be carried out by the user, and should look something like the following.

```
Module Proofs.
    Import Lemma_Defs.
    ...
```

```
4
5       Lemma Append_Nil_RList : ⊢ Append_Nil_RList_type.
6       Proof.
7           Admitted.
8
9       Lemma Append_Cons_RList : ⊢ Append_Cons_RList_type.
10      Proof.
11          Admitted.
12  End Proofs.
```

Finally, this instantiation can be explicitly checked against the lemma definitions given in `Lemma_Spec` using another Rocq module such as the following, and Rocq will complain if the proofs of the two lemmas supplied by `Proofs` are incomplete or fail to typecheck.

```
1  Module InstOK: CN_Lemmas.Gen_Spec.Lemma_Spec(Inst).
2
3    Module L := CN_Lemmas.Gen_Spec.Lemma_Defs (Inst).
4    Include Proofs.
5
6  End InstOK.
```

We intend to supply a template file containing these module definitions alongside our generated translations once this project is merged into the CN codebase. Given this setup, we successfully proved `Append_Nil_RList` and `Append_Cons_RList` by performing induction on the `List L1`. We display the proof of `Append_Nil_RList` below as an example of what a user-supplied proof for a Rocq lemma exported from CN actually looks like.

```
1  Lemma Append_Nil_RList : ⊢ Append_Nil_RList_type.
2  Proof.
3    iPureIntro.
4    intros L1 _.
5    induction L1.
6    + split; reflexivity.
7    + destruct IHL1.
8      split; auto; simpl.
9      rewrite H; auto.
10 Qed.
```

Here, `iPureIntro` is an Iris tactic that moves pure propositions from the Iris spatial context into the pure Rocq context, such that the entirety of Rocq's existing infrastructure can be used to prove them. Because every subterm of `Append_Nil_RList` is pure, the tactic rewrites the entire goal into the following pure proposition.

```
∀ x : List, (λ x0 : List, true → Append x0 Nil = x0 ∧ true) x
```

From here, we prove the statement by introducing the variables in the lemma statement,

performing induction on `L1`, then proving the two cases in a manner fairly typical to Rocq. We will omit the full proofs for the other lemmas discussed in this chapter. They are available for inspection in the codebase accompanying this report.

5.2 Case Study 2: Combining Iterated Ownership

We now move on to discuss how our implementation deals with resource ownership predicates, in particular iterated ownership predicates involving CN's `Each` keyword. Consider the following lemma about two different ways in which ownership of an integer array of length `n` with base pointer `p` can be asserted:

```
/*@
lemma each_lemma (pointer p, i32 n)
  requires
    take v = Owned<int>(p);
    take A = each(i32 j; 1i32 <= j && j < n)
              { Owned<int>(array_shift<int>(p,j)) };
    1i32 <= n; n <= MAXi32();
  ensures
    take A_post = each(i32 j; 0i32 <= j && j < n)
              { Owned<int>(array_shift<int>(p,j)) };
     A_post[0i32] == v;
@*/
```

This lemma states that given:

- Ownership of one integer value `v` at pointer `p`,
- Ownership of an `int` array segment `A` of length `n - 1` at the pointer obtained by array-shifting `p` by the size of one integer,
- Where `1i32 <= n`, such that the length of `A` is a non-negative integer, and `n <= MAXi32()`, such that `n` does not exceed the maximum value for 32-bit integers in C,

We should be able to obtain ownership of an `int` array `A_post` of length `n` at base pointer `p`, and the first element of this array should have value `v`. The translation of the lemma generated by our implementation is below:

```
Definition each_lemma_type : iProp Σ :=
  ∀ (p : Ptr), (n : Z),
    ∀ (v : Z), Owned_int p v -∗
    ∀ ( A : list Z),
      each_int (Z.to_nat (1)) (Z.to_nat (n) - (Z.to_nat (1)))%nat p A -∗
      ⌜ (2 <= n) ⌝ -∗
      ⌜ (n <= (2147483647)) ⌝ -∗
    ∃ ( A_post : list Z),
      each_int (Z.to_nat (0)) (Z.to_nat (n) - (Z.to_nat (0)))%nat p A_post
```

```
10                ∧ ⌜ ((nth (0) A_post 0) = v) ⌝
11                ∧ emp.
```

As discussed previously, both instances of `each_int` appearing in this Rocq definition are supplied an argument containing the difference between the upper and lower bounds of `j`, which gives the length of the desired array segment, as opposed to just the value of the upper bound `n`. A few other points of note include:

- Unsigned integers such as `n` and `j` in the original lemma are translated to `Z`, Rocq's type for unsigned integers, which need to be cast to Rocq's natural numbers `nat` when supplied as arguments to `each_int`. This does not cause problems because the pre-condition to the lemma enforces that all unsigned integers appearing in it are greater than or equal to zero.
- As discussed in Chapter 4, we translate arrays as Rocq lists. As a result, we translate the post-condition concerning the value of the first element of `A_post` using Rocq's `nth` function, which takes the index of the list value being accessed, the list itself, as well as a default value in case the index is out of bounds. We assume all value accesses are within bounds and set this final argument to zero by default.

Our proof for this lemma is fairly simple and involves instantiating the existential quantifier in the post-condition with the list `v :: A`, after which the goal can be easily discharged with the two ownership predicates given in the pre-condition as well as some rewriting of arithmetic operations.

5.3 Case Study 3: Imperative Queues

In this case study we consider imperatively-defined queues in C to demonstrate how our CN-to-Rocq translation deals with realistic user scenarios where resource predicates are needed. We divide this case study into 3 sections, which respectively address how imperative queues may be expressed on the specification level, the verification of a pop function, and the verification of a push function. This case study is sourced from the official CN tutorial, and we used our implementation to discover that one of the lemma statements used in it is not proveable without additional pre-conditions.

5.3.1 Imperative Queues in CN

In the following C definitions, a `queue` is represented as a singly-linked list of nodes and consists of two pointers: `front`, pointing to the first element of the queue, and `back`, pointing to the final element of the queue. Queue elements are `queue_cell` structs with two fields, the first being the value of that queue element, and the second being a pointer to the next element of the queue.

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

struct queue {
  struct queue_cell* front;
  struct queue_cell* back;
};
```

Assuming the availability of inductively-defined lists on the CN level as detailed in Chapter 1 of this report, the following inductive resource predicate can be used to specify CN-level lists where each element is associated with an owned memory cell in a manner that mirrors the definition of a `queue` given above. The arguments f and b are pointers to the first and final elements of a `List` L (e.g. its `front` and `back`):

$$
\begin{aligned}
&QueueAux(f, b, L) = \\
&\qquad (f = b \ \wedge L = Nil \wedge emp) \vee \\
&\qquad (f \neq b \wedge \exists\, F : struct\ queue\_cell.\ f \mapsto_{queue\_cell} H \star \\
&\qquad\quad (F.next \neq null \star \\
&\qquad\quad (\exists\, B :\ List.\ QueueAux(F.next, b) \ \star \\
&\qquad\qquad L = Cons(F.first, B)))))
\end{aligned}
$$

The predicate above holds if either:

- The pointers f and b are equal, in which case the input list must be the empty list (Nil) and the heap must be the empty heap (emp), or if
- The pointers f and b are not equal, in which case there must exist some pointee value F at f of type *struct queue_cell*, as well as some list B satisfying *QueueAux(F.next, b)*, representing the tail of L. The input list must also be equal to the list obtained by cons-ing $F.next$ onto B, and the pointer $F.next$ must not be a null pointer.

Together, the two clauses of the predicate traverse a *List* L and asserts ownership of a *queue_cell* containing each element of `L` (excluding the final element), as well as enforcing that L is equal to the list elements stored in the chain of *queue_cell*s starting from the pointee value of f. Therefore, it links our specification-level list `List` with the C-level `queue` defined earlier. The CN implementation of this predicate is the following:

```
/*@
predicate (datatype List) QueueAux (pointer f, pointer b) {
  if (ptr_eq(f,b)) {
    return Nil{};
```

```
  } else {
    take F = Owned<struct queue_cell>(f);
    assert (!is_null(F.next));
    take B = QueueAux(F.next, b);
    return Cons{Head: F.first, Tail: B};
  }
}
@*/
```

And the translation generated by our CN-to-Rocq translation mechanism is the following.

```
Fixpoint QueueAux (f : Ptr) (b : Ptr) (ret : List) {struct ret}  : iProp Σ :=
    (⌜ (f = b) ⌝  ∧  ⌜ ret = (Nil) ⌝) ∨
    (⌜ (Is_true true) ⌝ ∧ ⌜ ~(f = b) ⌝  ∧
    ∃ (F : queue_cell), Owned_queue_cell f F ⋆
      ⌜ (~ (F.(next) = 0)) ⌝ ⋆
      ⌜ ((F.(next) = b) ∨ (~ (F.(next) = b))) ⌝ ⋆
      ∃ (B : List), QueueAux F.(next) b B ⋆
      ⌜ ret = (Cons F.(first) B) ⌝).
```

We can then define two additional resource predicates `QueueFB` and `QueuePtr_At` to "wrap" `QueueAux`, through which we derive a specification-level definition of imperative queues. The definition of `QueuePtr_At` is given below.

```
/*@
predicate (datatype List) QueuePtr_At (pointer q) {
  take Q = Owned<struct queue>(q);
  assert (   (is_null(Q.front)  && is_null(Q.back))
           (!is_null(Q.front) && !is_null(Q.back)));
  take L = QueueFB(Q.front, Q.back);
  return L;
}
@*/
```

Here, `QueuePtr_At` asserts ownership of a `struct queue Q` at pointer `q`, and checks that the `queue` is well-defined (e.g. it is either empty, in which both its `front` and `back` should be null pointers, or non-empty, in which case neither should be null pointers). It then calls `QueueFB`, which asserts ownership of the final `queue_cell` of `Q` and calls `Queue_Aux` on those `queue_cell`s between the first and final members of `Q`. We omit the definition of `QueueFB` because it is not particularly interesting.

Given this setup, an assertion such as

```
take Q = QueuePtr_At(q)
```

binds `Q` to a `List` for which there exists a corresponding `struct queue` with identical elements, and asserts that we have ownership of both this `struct queue` as well as the

`struct queue_cell`s it comprises of.

5.3.2 Pop

We may now define `pop_queue`, a C function that removes an element from the front of a non-empty queue. The definition below takes `q`, a pointer to a `queue` as input, and compares its `front` and `back` fields: if the two are equal, the `queue` must only contain a single element, in which case we return the `int` value stored `h`, the only `queue_cell` in our input `queue`. We then free all memory involved in that `queue_cell`. If `front` and `back` are not equal, we still return the `int` value stored in `h`, but we also set the `front` of `q` to be the pointer to the next `queue_cell`, representing the removal of `h` from `q`. We then free the memory used for `h`.

```
1  int pop_queue (struct queue *q)
2  {
3    struct queue_cell* h = q->front;
4    if (h == q->back) {
5      int x = h->first;
6      free_queue_cell(h);
7      q->front = 0;
8      q->back = 0;
9      return x;
10   } else {
11     int x = h->first;
12     q->front = h->next;
13     free_queue_cell(h);
14     return x;
15   }
16 }
```

Now, suppose we wished to verify that `pop_queue` satisfies the following properties:

- Before executing `pop_queue`, we possess ownership of all memory cells used in its input `queue`, and
- The input queue `Q` is not empty.
- After executing `pop_queue`, we return ownership of all memory cells used by the `queue`, minus the `queue_cell` we free during the popping. The `queue` stored in these memory cells should be equivalent to the tail of the input `queue`.
- The return value of the `pop_queue` should be equal to the head of the input `queue`.

The following annotated version of `pop_queue` expresses these properties through `requires` and `ensures` clauses following function declaration. The next two annotations making use of the functions `split_case` and `unfold` both invoke CN's built-in tactics/proof hints. They respectively instruct CN with the appropriate guarded case at which it should un-

fold the definition of the `QueuePtr_At` predicate, and to unfold the recursive definition of the `Snoc` function. The final remaining annotation invokes a lemma called `pop_lemma`.

```
int pop_queue (struct queue *q)
/*@ requires take Q = QueuePtr_At(q);
             Q != Nil{};
    ensures take Q_post = QueuePtr_At(q);
            Q_post == Tl(Q);
            return == Hd(Q);
@*/
{
  /*@ split_case is_null(q->front); @*/
  struct queue_cell* h = q->front;
  if (h == q->back) {
    int x = h->first;
    free_queue_cell(h);
    q->front = 0;
    q->back = 0;
    /*@ unfold Snoc(Nil{}, x); @*/
    return x;
  } else {
    int x = h->first;
    /*@ apply pop_lemma(h->next, q->back, x); @*/
    q->front = h->next;
    free_queue_cell(h);
    return x;
  }
}
```

The lemma `pop_lemma` states an obvious property of the `Snoc` function: given an owned `queue Q`, an additional owned `queue_cell B`, and an arbitrary integer `x`, appending `B` to the end of `Cons x Q` using `Snoc` should yield a new list whose head is still `x`, and whose tail is `Snoc (Q,B.first)` (we recall that `first` is the `queue_cell` field containing the queue element stored there, while `next` contains a pointer to the `queue_cell` after it). It is stated as a lemma because proving it requires performing induction on `Q`.

```
/*@
lemma pop_lemma (pointer front, pointer back, i32 x)
  requires
      take Q = QueueAux(front, back);
      take B = Owned<struct queue_cell>(back);
  ensures
      take Q_post = QueueAux(front, back);
      take B_post = Owned<struct queue_cell>(back);
      Q == Q_post; B == B_post;
      let L = Snoc (Cons{Head: x, Tail: Q}, B.first);
      Hd(L) == x;
```

```
      Tl(L) == Snoc (Q, B.first);
@*/
```

The following is the Rocq translation we generate for `pop_lemma`.

```
Definition pop_lemma : iProp Σ :=
    ∀ (front : Ptr), ∀ (back : Ptr), ∀ (x : Z),
      ∀ (Q : List), QueueAux front back Q  -∗
      ∀ (B : queue_cell), Owned_queue_cell back B  -∗
      ∃ (Q_post : List), QueueAux front back Q_post ∗
      ∃ (B_post : queue_cell), Owned_queue_cell back B_post ∧
        ⌜ (Q = Q_post) ⌝ ∧ ⌜ (B = B_post) ⌝ ∧
        (let L := (Snoc (Cons x Q) B.(first)) in
          ⌜ ((Hd L) = x) ⌝ ∧
          ⌜((Tl L) = (Snoc Q B.(first))) ⌝ ∧
          emp).
```

We were able to successfully prove the Rocq translation of `pop_lemma` given above within Rocq, with minimal effort - the proof only required inducting on `Q`, instantiating the existentials, and eliminating connectives. We will see that this is not always the case in the next part of this section.

5.3.3 Push

Now, we consider `push_queue`, a C function that pushes a new element to the back of a `queue`. This function takes an `int x` and pointer `q` to a `struct queue` as input. It then allocates memory for a new `queue_cell` called `c`, sets `x` as the queue element stored in `c`, and sets the pointer to the next node of `c` to be the null pointer. Following this, it checks if the input `queue` is empty, in which case it sets the pointee value of `q` as `c`, yielding a singleton `queue`, else setting its final `queue_cell` to `c`, yielding a `queue` with an additional element `x` at its back.

The annotations to the function check that the predicate `QueuePtr_At` holds for the input `queue` before and after `push_queue` is executed, and that the elements of the post-push `queue` are equal to those of the list obtained by `Snoc`-ing `x` to the elements of the original `queue`. A lemma `push_lemma` is invoked to facilitate CN's reasoning.

```
void push_queue (int x, struct queue *q)
/*@ requires take Q = QueuePtr_At(q);
    ensures take Q_post = QueuePtr_At(q);
            Q_post == Snoc (Q, x);
@*/
{
  struct queue_cell *c = malloc_queue_cell();
  c->first = x;
  c->next = 0;
```

```
  if (q->back == 0) {
    q->front = c;
    q->back = c;
    return;
  } else {
    struct queue_cell *oldback = q->back;
    q->back->next = c;
    q->back = c;
    /*@ apply push_lemma (q->front, oldback); @*/
    return;
  }
}
```

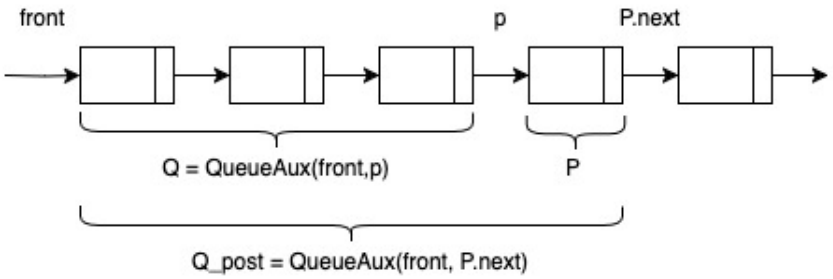
The lemma `push_lemma` states that given n linked `queue_cell`s, for which we have ownership of the first $n-1$ cells as a `queue` through the `QueueAux` predicate defined earlier in this chapter, as well as separate ownership of the final cell, we should be able to obtain ownership of all n cells collectively through `QueueAux`. We also have that the `List` representing all n linked `queue_cell`s should be equal to the result of `Snoc`-ing the value at the final `queue_cell` to the values in the first $n-1$ `queue_cell`s.

```
/*@
lemma push_lemma (pointer front, pointer p)
  requires
      ptr_eq(front, p) || !addr_eq(front, p);
      take Q = QueueAux(front, p);
      take P = Owned<struct queue_cell>(p);
      !is_null(P.next);
  ensures
      ptr_eq(front, P.next) || !addr_eq(front, P.next);
      take Q_post = QueueAux(front, P.next);
      Q_post == Snoc(Q, P.first);
@*/
```

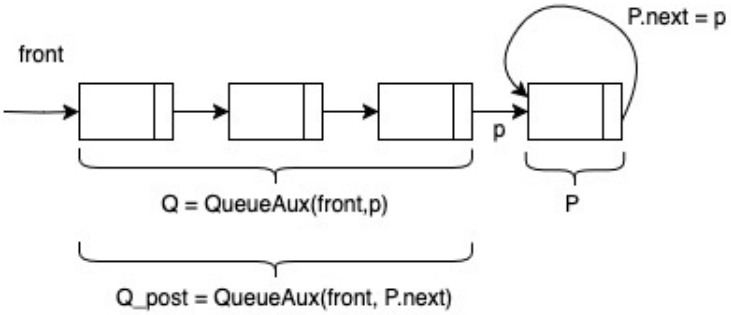
While `push_lemma` seems believable, the following un-provable proof obligation arose during our attempt to prove the lemma in Rocq:

$$\ulcorner \texttt{p} \neq \texttt{D.next P} \urcorner$$

To illustrate why this is unprovable, consider the following graphical representation of a `queue`. We represent `queue_cell`s with boxes and annotate them with curly brackets to illustrate which cells are included in each of the resource ownership predicates appearing in `push_lemma`, and the post-condition `Q_post == Snoc(Q,P.first)` looks fairly convincing when the `queue` is drawn like this: the sequence of queue-cells owned in `Q_post` seems to be exactly the ones obtained by `Snoc`-ing `P` to the cells owned by `Q`.



The proof obligation ⌜`p ≠ D.next P`⌝ asks us to rule out the scenario where the pointer stored at the `next` field of `P` is also `p`. More specifically, because the inductive resource predicate `QueueAux` returns an empty list when its two input pointers are equal, `QueueAux(front,P.next)` will not assert ownership for the `queue_cell P` when `p = P.next` and renders the lemma's conclusion `Q_post == Snoc(Q,P.first)` untrue, as illustrated below, where `Q_post` clearly does not contain the cell `P`. The lemma's assumptions are not strong enough to preclude this situation.

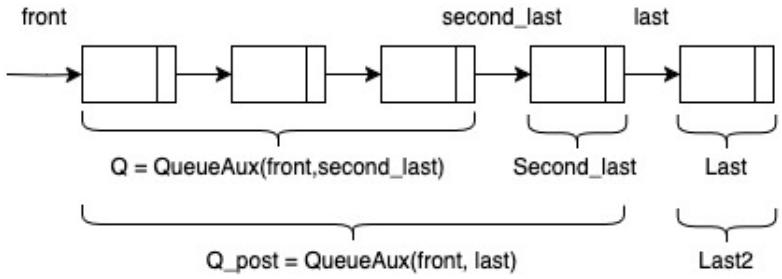


Upon reporting the problem described above to CN maintainers, they supplied us with a new version of the lemma, given below. This lemma states that given three pointers `front`, `second_last`, and `last` pointing to the first, second-last, and last `queue_cells` of a `queue`, as well as ownership of the `queue Q` between `first` and `second_last`, and separate ownership of the cells at `second_last` and `last`, one can view all `queue_cells` between `first` and `last` as an owned `queue` through `QueueAux`. This owned `queue` should have elements equal to that of the list obtained by `Snoc`-ing `Q` to `Second_last.first`, and we should continue to have ownership of the `queue_cell` at `last`. If this is confusing, there is an accompanying illustration under the lemma statement.

```
1  /*@
2  lemma push_lemma_2 (pointer front,
3                      pointer second_last,
4                      pointer last)
5    requires
6        ptr_eq(front, second_last) || !addr_eq(front, second_last);
7        take Q = QueueAux(front, second_last);
8        take Second_last = Owned<struct queue_cell>(second_last);
9        ptr_eq(Second_last.next, last);
10       take Last = Owned<struct queue_cell>(last);
```

```
11    ensures
12        ptr_eq(front, last) || !addr_eq(front, last);
13        take Q_post = QueueAux(front, last);
14        take Last2 = Owned<struct queue_cell>(last);
15        Q_post == Snoc(Q, Second_last.first);
16        Last == Last2;
17  @*/
```

The following diagram illustrates the lemma statement: we begin with ownership of the `queue_cell`s in `Q`, as well as `Second_last` and `Last`. From this, we should be able to show that `QueueAux(front,last)` holds, while retaining separate ownership of the `queue_cell` at `last`.



To see why this lemma avoids the loopy queues that render the previous lemma unprovable, we can consider the pointer `Second_last` as this lemma's analogue of the pointer `p` from the previous lemma, while `last` is the analogue of `P.next`. Then, separate ownership of two `queue_cell`s `Second_last` and `Last` at the pointers `second_last` and `last` implies `second_last` $\neq$ `last`. As `Q` asserts ownership for all `queue_cell`s from `front` to `second_last`, this means no two pointers occurring across the queue segment specified by `QueueAux(front,last)` are equal, eliminating the possibility of loopiness within that segment, which makes proving our new lemma possible.

Indeed, we succeeded in proving `push_lemma_2` but with some caveats. Firstly, the proof is more than 250 lines long and involved 6 subsidiary lemmas about various properties of the `QueueAux` predicate. Secondly, submitting a complete proof involving `QueueAux` to Rocq using the `Qed` keyword triggers an error about the well-formed-ness of `QueueAux`, so these lemmas must be submitted with the `Admitted` keyword even though their proofs are complete. The errors are caused by the fact that we disable Rocq's termination checker for translated resource predicates, as discussed in Section 4.3.3. Nonetheless, we are confident our resource predicates do terminate, which is guaranteed at a previous stage of the CN pipeline, so that our proofs are correct.

5.4 Comparison With Existing Implementation

We now iterate the specifications we set out for our implementation in Section 4.1.

- Our implementation does not currently support CN-to-Rocq translations for *all* CN

language constructs. This is the only criteria our implementation fails on.

- Our implementation does contain Rocq translations of the full global typing context for a lemma, including features like recursive functions and resource predicates that were previously unsupported.
- Our implementation is accompanied by two libraries to facilitate the translation process as well as user-supplied proofs, one of which is new and contains the Iris infrastructure required for proving lemmas about resources to be possible.
- Our implementation generates a file that typechecks in Rocq, with a module structure explicitly set up to check user-supplied proofs against lemma statements. We credit the existing implementation for already having these features in place.

Compared to the existing implementation, we achieve one additional specification criteria while getting significantly closer to reaching the final remaining one.

5.5 Limitations

We now discuss the limitations of our implementation. A potential area of improvement is to use a Rocq bitvector library to represent bitvectors, as opposed to our current strategy of using integers. This would entail finding suitable bitvector representations for CN base-types, as well as re-implementing some of the built-in ownership predicates in `CN_Lib_Iris`. With this addition, we will be able to support proofs that involve reasoning about bitvector operations.

A second limitation is that we lack full support for iterated resource predicates involving the `each` keyword in CN, as we currently only support translating `each` predicates representing ownership of arrays. While this covers the primary use case for `each` predicates, generalizing support to iterated resource ownership predicates where this iteration can take any form will still be an improvement. Additionally, translation support for maps will allow us to translate array ownership predicates more faithfully because arrays are implemented as maps from indices to values in CN.

Finally, we currently lack a proof of termination for inductive resource predicates in Rocq. Supplying such a proof will be a large undertaking but it is morally good to have and we expect it will make a number of Iris proofs involving heavy work with inductive resource predicates more pleasant to write.

We note that our lack of support for some CN basetypes is a less pressing concern as adding them would entail constructing our own Iris resource algebra to reflect the memory model CN works with, which is still a work in progress further upstream.

Chapter 6

Summary and conclusions

To summarise, this project makes significant extensions to the CN-to-Rocq translation mechanism currently employed by the CN verification tool. Our implementation successfully provides several important new features for CN users, including translation support for recursive functions, inductive resource predicates, and lemmas involving resource ownership. The bulk of our work was dedicated towards setting up the Iris infrastructure necessary for reasoning about lemmas involving resources, and additional features will be significantly easier to implement in the future due to the significant codebase refactoring effort conducted as part of this project. As a consequence of the wider subset of the CN specification language supported by our translation, we discovered an error in an example employed by the official CN documentation. We plan to deploy our implementation to the CN codebase once code reviews are complete.

There are many possibilities for future work, with the obvious option being the lifting of the limitations addressed in Section 5.4. Additionally, we propose that the Rocq libraries accompanying our implementation can be extended with tactics dealing with some of the repetitive scenarios that occurred during our case studies, alongside a proof of termination for our translated inductive resource predicates. A long-term goal would be replacing the simple resource algebra used in our Iris instantiation with one that reflects a more realistic memory model, such as the VIP model [19], which CN is cucrently experimenting with. This would entail constructing a new Iris resource algebra that accurate reflects the chosen memory model, which will be a very non-trivial amount of work.

Bibliography

[1] Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Rini Banerjee, Kayvan Memarian, Dhruv Makwana, Christopher Pulte, Neel Krishnaswami, and Peter Sewell. Fulminate: Testing cn separation-logic specifications in c. *Proc. ACM Program. Lang.*, 9(POPL), January 2025.

[3] Lars Birkedal and Ale˘s Bizjak. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. `https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf`, 2023.

[4] Allan Blanchard. ntroduction to C program proof with Frama-C and its WP plugin. `https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf/`, 2020.

[5] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 259–270, New York, NY, USA, 2005. Association for Computing Machinery.

[6] R Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, (7):23–50, 1972.

[7] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 366–378, 2007.

[8] Tej Chajed. Heaplang overview.

[9] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, page 161–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] Anthony Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. Verified compilation of cakeml to multiple machine-code targets. In *Proceedings of the*

6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, page 125–137, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: an extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 653–669, USA, 2016. USENIX Association.

[12] Gernot Heiser, Gerwin Klein, and June Andronick. sel4 in australia: from research to real-world trustworthy systems. *Commun. ACM*, 63(4):72–75, March 2020.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[14] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. *SIGPLAN Not.*, 36(3):14–26, January 2001.

[15] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: a powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, page 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[17] RALF JUNG, ROBBERT KREBBERS, JACQUES-HENRI JOURDAN, ALEŠ BIZJAK, LARS BIRKEDAL, and DEREK DREYER. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[18] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. *SIGPLAN Not.*, 52(1):205–217, January 2017.

[19] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. Vip: verifying real-world c idioms with integer-pointer casts. *Proc. ACM Program. Lang.*, 6(POPL), January 2022.

[20] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

[21] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally verified memory protection for a commodity multiprocessor hypervisor. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3953–3970. USENIX Association, August 2021.

[22] Gregory Malecha, Gordon Stewart, František Farka, Jasper Haag, and Yoichi Hirai. Developing with formal methods at bedrock systems, inc. *IEEE Security Privacy*, 20(3):33–42, 2022.

[23] K. Memarian. *The Cerberus C Semantics*. University of Cambridge, 2022.

[24] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring c semantics and pointer provenance. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[25] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N.M. Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2016. PLDI 2016 Distinguished Paper award.

[26] Peter O'Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, January 2019.

[27] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic*, pages 1–19, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[28] Peter W. O'Hearn. Resource interpretations, bunched implications and the $\alpha\lambda$-calculus (preliminary version). In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, pages 258–279, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[29] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *The Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[30] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. Cn: Verifying systems c code with separation-logic refinement types. *Proc. ACM Program. Lang.*, 7(POPL), January 2023.

[31] Christopher Pulte, Benjamin C. Pierce, Cole Schlesinger, Jessica Shi, and Elizabeth Austell. CN Tutorial. `https://rems-project.github.io/cn-tutorial/`, 2025.

[32] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[33] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: automating the foundational verification of c code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 158–174, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Julien Signoles, Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, and Boris Yakobowski. Frama-c: a software analysis perspective. volume 27, 10 2012.

[35] David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[36] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 866–881, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Amin Timany and Lars Birkedal. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.