Ph.D. DISSERTATION

End-to-End Verification Supporting Integer-Pointer Casting

정수-포인터 변환을 포함한 프로그램을 처음부터 끝까지 검증하기

February 2025

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Yonghyun Kim

End-to-End Verification Supporting Integer-Pointer Casting

정수-포인터 변환을 포함한 프로그램을 처음부터 끝까지 검증하기

지도교수 허 충 길

이 논문을 공학박사학위논문으로 제출함

2024 년 12 월

서울대학교 대학원

컴퓨터공학부

김 용 현

김 용 현의 공학박사 학위논문을 인준함

2024 년 12 월

위 원 장	이 광 근	(인)
부위원장	허 충 길	(인)
위 원	강 지 훈	(인)
위 원	김 지 응	(인)
위 원	이 재 욱	(인)

Abstract

This thesis presents an end-to-end verification method for C programs containing integer-pointer casts. While previous approaches have developed formal memory models supporting integer-pointer casts, each has limitations preventing complete end-to-end verification: they are not designed for end-to-end verification. These approaches either fail to support important source-level coding patterns, cannot justify certain compilation passes, or lack a source-level logic for program verification.

This thesis introduces Archmage, a framework designed for end-to-end verification of programs containing integer-pointer casts. Archmage supports a wide range of source-level coding patterns, backend optimizations, and provides a formal source-level logic for verification. Within the Archmage framework, we present two systems: CompCertCast, an extension of CompCert that provides a fully verified compilation chain for programs containing integer-pointer casts, and Archmage logic, a source-level logic for reasoning about integer-pointer casts. We demonstrate the effectiveness of our approach by minimizing the overhead of formally supporting integer-pointer casts in CompCertCast, and by verifying examples including an xor-based linked-list.

Keywords: Integer-Pointer Casting, Compiler, CompCert, Separation Logic, Coq, Formal Verification

Student Number: 2017-22945

Contents

List of Figures

Chapter 1

Prologue

1.1 Introduction

Integer-to-pointer casting is a common technique used in low-level C system programming, making the verification of such programs an important challenge. While various formal methods supporting integer-pointer casts have been proposed, existing approaches have limitations in end-to-end verification: they either fail to support important source-level coding patterns [1, 2], cannot justify certain compiler optimizations [3], or lack source-level methods for program verification [4, 5].

In this thesis, we present Archmage, a framework that enables end-to-end verification of C programs containing integer-pointer casts. The framework addresses two key aspects of end-to-end verification: compiler verification and program verification. The first contribution, presented in §5, is CompCertCast, a verified C compiler supporting integer-pointer casts. Built as an extension of CompCert, it ensures the correct compilation of C programs containing integer-

pointer casts from source code to assembly level. The second contribution, presented in §6, is Archmage logic, a program logic for verifying C programs with integer-pointer casts. This logic extends conventional separation logic and is sufficiently expressive to verify complex implementations, such as XOR-based doubly linked lists. Both systems are founded on a unified memory model called Archmage (§4), specifically designed for end-to-end verification.

This thesis draws heavily on the work and writing in the following paper: [6]

The background on CompCert, a formally verified C compiler that serves as our foundation, is described in detail §1.2.

1.2 Background

This section provides helpful background knowledge for understanding the concepts presented in this thesis. Since Archmage builds upon CompCert for its compiler verification we focus on CompCert's core principles and architecture. Readers who are already familiar with CompCert's proof techniques and memory model may skip to the next section.

1.2.1 CompCert

CompCert, the first verified C compiler, has been adopted by several end-to-end verification projects [2, 7, 8] for compiler verification. In this subsection, we first provide a high-level overview of CompCert and explain the techniques used to prove compilation correctness. We then examine CompCert's Logical memory model, which has influenced many C formal memory models.

CompCert's Architecture

CompCert compiles C source code into assembly code. During compilation, CompCert uses 8 intermediate languages and executes 20 compilation passes. CompCert's compilation passes include important optimizations commonly found in modern compilers, such as dead code elimination, function inlining, and register allocation. Each compilation pass has been proven to be "correct". The following paragraphs explain the definition of compiler correctness and the techniques CompCert uses to prove it.

Compiler Correctness. Compiler correctness in CompCert (and many other compiler verification approaches) is defined via the concept of *behavioral refinement*. That is, CompCert states that a compiler is "correct" if a source program SRC compiles into a target program TGT, and the set of behaviors of the target (denoted as Beh(TGT)) is a subset of the behaviors of the source (Beh(SRC)). A program's behavior refers to the trace—either finite or infinite—of I/O events, such as system calls, that occur during program execution.

CompCert establishes behavioral refinement through showing *simulation* holds between the source SRC and the target TGT. Specifically, CompCert employs two types of simulations: *forwards* simulation and *backwards* simulation. Intuitively, when interpreting both SRC and TGT as state transition systems, a *forwards* simulation states for each execution step in the source program (SRC), the target program (TGT) has corresponding steps it can take while triggering the same event. More formally, this concept is captured in Eqn. (1.1) for source-level states st_{SRC} and st'_{SRC}, target-level states st_{TGT} and st'_{TGT}, a relation R, an event ev, and silent steps τ:

$$\forall (st_{\mathsf{SRC}}, st_{\mathsf{TGT}}) \in R,\ \forall ev, st'_{\mathsf{SRC}},\ \ st_{\mathsf{SRC}} \overset{ev}{\hookrightarrow} st'_{\mathsf{SRC}} \implies \\ \exists st'_{\mathsf{TGT}},\ \ st_{\mathsf{TGT}} \overset{\tau}{\hookrightarrow}^* \overset{ev}{\hookrightarrow} \overset{\tau}{\hookrightarrow}^* st'_{\mathsf{TGT}} \wedge (st'_{\mathsf{SRC}}, st'_{\mathsf{TGT}}) \in R\ . \tag{1.1}$$

Similarly, *Backwards* simulation also relates execution steps between source and target programs, but establishes that for each execution step in the target program (TGT), the source program (SRC) must be able to take corresponding steps while triggering the same event. The formal definition of *backwards* simulation is presented in Eqn. (1.2).

$$\forall (st_{\mathsf{SRC}}, st_{\mathsf{TGT}}) \in R,\ \forall ev, st'_{\mathsf{TGT}},\ \ st_{\mathsf{SRC}} \overset{ev}{\hookrightarrow} st'_{\mathsf{TGT}} \implies \\ \exists st'_{\mathsf{SRC}},\ \ st_{\mathsf{SRC}} \overset{\tau}{\hookrightarrow}^* \overset{ev}{\hookrightarrow} \overset{\tau}{\hookrightarrow}^* st'_{\mathsf{SRC}} \wedge (st'_{\mathsf{SRC}}, st'_{\mathsf{TGT}}) \in R\ . \tag{1.2}$$

Forwards simulation provides a more practical approach to compiler verification, though it guarantees behavioral refinement only when the target language is deterministic. [1] CompCert primarily employs *forwards* simulation in its verification process, and §5 of this dissertation will discuss alternative proof techniques that can serve as replacements for forward simulation.

The Logical Memory Model of CompCert

The memory model serves as a central component of CompCert's architecture. A distinctive feature of CompCert is that all of its languages—C, Assembly, and the intermediate languages—share the same memory model. Therefore, understanding this memory model is essential for comprehending CompCert's features. Using fig. 1.1 and fig. 1.2, we will examine CompCert's logical memory model in detail.

Definition of the Memory Model. In CompCert's logical memory model, memory (Mem) consists of a finite set of memory blocks, each identified by a unique

[1] More precisely, this requires a receptive source and determinate [9] target language.

$M \in \mathsf{Mem} \stackrel{\text{def}}{=} \mathsf{BlockID} \xrightarrow{fin} \mathsf{Block}$
$b \in \mathsf{BlockID} \stackrel{\text{def}}{=} \mathbb{N} \quad sz \in \mathsf{Size} \stackrel{\text{def}}{=} \mathbb{N} \quad v \in \mathsf{Val} \stackrel{\text{def}}{=} \mathsf{Int} \uplus \mathsf{Pointer} \uplus \{undef\}$
$\mathsf{Block} \stackrel{\text{def}}{=} \{\ (live, sz, c) \mid live \in \mathbb{B} \wedge sz \in \mathsf{Size} \wedge c \in \mathsf{Val}^{sz}\ \}$
$(b, ofs) \in \mathsf{Pointer} \stackrel{\text{def}}{=} \mathsf{BlockID} \times \mathsf{Int}$

Figure 1.1: Type definitions for CompCert Logical Memory.

identifier (BlockID). This feature is captured in fig. 1.1 as partial function $\mathsf{Mem} \stackrel{\text{def}}{=} \mathsf{BlockID} \xrightarrow{fin} \mathsf{Block}$. Each block in the logical memory model corresponds to an actual memory object, which is defined by an array of values and a liveness Boolean flag (*live*). The *live* flag indicates whether the block is accessible or freed. *Values* in CompCert are represented as integers, pointers, or undefined values. While CompCert also supports floating-point numbers, we exclude these from our discussion for simplicity. A pointer in CompCert consists of a pair (b, ofs), where b is a block identifier and *ofs* is an offset within block b." While CompCert's logical memory model provides useful properties for verifying compiler optimization correctness (such as ensuring functions have exclusive ownership of non-escaped blocks), it has limitations in handling integer-pointer casting. CompCert implements integer to pointer casting as a simple identity function, rather than generating an integer value that properly corresponds to a pointer value (b, ofs). Although integer variables in CompCert can store pointer values, CompCert does not support various operations on these pointer-containing integers, such as bitwise-XOR and division. A detailed discussion of formal memory models that address these limitations in integer-pointer casting can be found in §7.

Remark. *For simplicity of presentation, in this section, we will assume that values are of size 1, which means that the size of a block is equivalent to the number of values it can store, and that values are not subject to certain alignment constraints that are present in the C standard (e.g., in C, 32-bit integer*

$$\frac{b \text{ is fresh} \quad \text{blk} = (\mathit{true}, sz, \mathit{undef}^{sz})}{(\mathsf{alloc}(sz), M) \to ((b, 0), M[b \mapsto \text{blk}])} \qquad \frac{M(b) = (\mathit{true}, sz, c)}{(\mathsf{free}((b, 0)), M) \to ((), M[b \mapsto (\mathit{false}, sz, c)])}$$

$$\frac{M(b) = (\mathit{true}, sz, c) \quad c' = c[\mathit{ofs} \mapsto v]}{(\mathsf{store}((b, \mathit{ofs}), M) \to ((), M[b \mapsto (\mathit{true}, sz, c')])} \qquad \frac{M(b) = (\mathit{true}, sz, c) \quad c[\mathit{ofs}] = v}{(\mathsf{load}((b, \mathit{ofs})), M) \to (v, M)}$$

$$\frac{}{(\mathsf{ptoi}(v), M) \to (v, M)} \qquad \frac{}{(\mathsf{itop}(v), M) \to (v, M)}$$

Figure 1.2: Selected and simplified rules for the semantics of Logical Memory.

values must be aligned to 4-bytes). Additionally, while CompCert's memory access permissions for live memory areas have four variations (freeable, writable, readable, and Nonempty), we simplified permissions to just a block liveness flag.

Operational Semantics of Memory Operations. Having examined CompCert's memory representation, we now turn to the operational semantics of memory operations as defined in fig. 1.2.

The memory model defines four fundamental operations: (*i*) alloc operation creates a new block with a fresh block identifier b, setting its size to sz and initializing all contents with undefined values; (*ii*) free operation checks that a block is freeable by checking the premise $M(b) = (\mathit{true}, sz, c)$, then sets the block's liveness flag to false when the check succeeds; (*iii*) store operation first ensures a block is writable by checking the premise $M(b) = (\mathit{true}, sz, c)$, then rewrites the *ofs*-th content of c to value v, producing c' in the rule's conclusion; (*iv*) load operation checks block readability through the premise $M(b) = (\mathit{true}, sz, c)$, then reads the *ofs*-th content of c. As mentioned earlier, CompCert implements integer-pointer casting as identity functions, as shown in the final two rules of fig. 1.2.

Chapter 2

Introduction: Towards End-to-End Verification Supporting Integer-Pointer Casting

Pointers have long been a key feature of the C programming language. In particular, C supports the idea of *integer-pointer casting*, which allows one to cast a pointer p an integer i, which represents the concrete address in which p resides in memory (and vice versa). This feature brings with it the advantage that it extends the wide variety of operations that are supported on integers—for example, bitwise operations such as bitmasks—towards pointers, for which it is difficult to define such operations otherwise, giving programmers much more power when manipulating pointers in their code. Integer-pointer casting is now a critical feature of C, used in many coding patterns such as pointer hashing, efficient linked list implementations, or tagged pointers, which in turn provide time- and space-efficient implementations of widely used algorithms and data structures.

Given the prevalence of integer-pointer casting in C code, a formal reasoning scheme supporting integer-pointer casting is clearly desirable. Ideally, such a formal scheme should satisfy two major desiderata: (*i*) the scheme should facilitate *source-level verification* on C code (e.g., in the form of pre- and postconditions) and (*ii*) the scheme should be compatible with a *verified compiler*—that is, CompCert [1]—allowing the compiler to establish the soundness of compilation and optimizations on code that contains integer-pointer casts. We argue that these desiderata are essential for bringing verification on integer-pointer casts up to par with other more well-studied features of the C language, which enjoy the benefits of end-to-end verification.

However, it turns out that developing a reasoning scheme for integer-pointer casts satisfying the aforementioned desiderata is a surprisingly complex task, requiring careful consideration of the interplay of integer-pointer casting with other features of the C language. Part of this conundrum is due to the fact that the basic view of memory as defined by the C standard is actually comprised of fully abstract *logical blocks* that have nothing to do with integers, eschewing the intuition that pointers are integers that represent physical addresses in memory. The C standard then gives a list of rules that pointers and the results of integer-pointer casts should adhere to—a list that is sadly, as is standard of the C standard, written in prose, and therefore very difficult to translate fully to into a formal semantics. For this reason, CompCert is markedly limited in its ability to support code that contains integer-pointer casts. For example, CompCert does not support bitwise operations on integers resulting from a pointer-to-integer cast, which in turn makes it impossible for CompCert to support some commonly used C patterns such as pointer hashing.

There have of course been many previous attempts [4, 5, 10, 11, 12] to formalize and verify integer-pointer casting outside of CompCert as well. For example,

PNVI-ae-udi [5] represents a significant attempt at formalizing various features of the C standard, including integer-pointer casts, that is successful enough for consideration to be added as part of the C standard. The Quasi-Concrete model [4] supports a wide range of C idioms and compiler optimizations, and is formalized in Coq. While not an example of formalizing C semantics, the Twin-Allocation model [11] was developed in order to semi-formally justify pointer-related optimizations that are performed by LLVM [13], and thus provides a good formal explanation of how integer-pointer casts should operate within the compilation pipeline.

Unfortunately, despite each of these approaches having their own unique advantages, they (aside from perhaps the Quasi-Concrete model) share a limitation in that they all use their own separate representation of memory. This makes it difficult to merge these approaches with CompCert as to obtain a full verified compiler that is capable of establishing the soundness of both code that does and does not contain integer-pointer casts. Thus, for example, while VIP [10] may be able to provide powerful source-level guarantees on certain commonly used coding patterns, it is also incapable of preserving these guarantees throughout the compilation pipeline—a feature that is clearly desirable, but difficult to achieve without integration with CompCert.

In this paper, we introduce the Archmage framework, an end-to-end verification framework for programs with integer-pointer casts. Archmage framework is based on the eponymous memory model Archmage, a new memory model that is designed to couple tightly with CompCert and thus facilitate end-to-end verification of programs containing integer-pointer casts. Archmage strives to support as many source-level features of C as possible, and allow CompCert to correctly perform as many optimizations on these features. To make point of these claims, we provide a concrete implementation of CompCert extended

with Archmage, named CompCertCast (§5): an actual verified compiler that is engineered to preserve as many backend optimizations performed by the original CompCert as possible. In addition to allowing CompCert to compile and optimize code containing integer-pointer casts, CompCertCast also *extends* certain optimizations towards code containing non-trivial integer-pointer casts; this allows CompCertCast to minimize the performance gap that is inevitable from considering integer-pointer casts and out-of-memory in a formal manner, furthering the practicality of Archmage as a basis for end-to-end verification.

In addition to the tight integration with CompCert as shown by CompCertCast, the Archmage framework also provides a *source-level separation logic* for verifiying properties of programs that contain integer-pointer casts (§6). Archmage logic is designed with ease-of-use for the end user in mind, while at the same time supporting a wide range of commonly used coding patterns thanks to the generality of Archmage. This unique combination of source-level support and usability makes Archmage logic an ideal choice for source-level verification of programs with complex pointer manipulation. As an example, in §6, we rely on Archmage logic to prove the correctness of an XOR-based linked list implementation: Archmage represents the first end-to-end verification of an xor-list implementation.

Contributions. To summarize, this paper makes the following contributions:

- Archmage, the first memory model designed to facilitate end-to-end verification (§4).
- CompCertCast, a faithful extension of CompCert to work with Archmage, bringing verified compilation to integer-pointer casting and, when combined with Archmage logic, brings true end-to-end verification to programs containing integer-pointer casts (§5).

- Archmage logic, a source-level separation-style logic for built on top of Archmage, that allows users to easily write and prove properties about programs, while supporting a wide range of C features used in practice (§6).

§3 provides a high-level overview of the entire Archmage framework, with a focus on new contributions. §7 discusses related work; and §8 concludes. Archmage, Archmage logic, and CompCertCast are all implemented using the Coq proof assistant [14]. Together, they provide the first full realization of a end-to-end verification pipeline with support for integer-pointer casts.

Chapter 3

Overview of Contributions

In this section, we provide an overview of the three main components of our system: (*i*) the memory model Archmage, (*ii*) CompCertCast, the integration of Archmage with CompCert, and (*iii*) Archmage logic, a source-level separation-style logic built on top of Archmage, with an emphasis on the benefits that each of the three components bring in comparison to existing work.

3.1 The Memory Model Archmage

Archmage is a memory model developed with the goal of enabling end-to-end verification for programs containing integer-pointer casts integration with CompCert. Archmage draws upon many concepts that were established in previous memory models [4, 11, 5, 10, 12], and in particular has many similarities to the Quasi-Concrete model [4]. However, there are also several key differences to Archmage that allow Archmage to capture a wider range of source-level coding patterns. We illustrate these differences through a coding pattern that is well-known to be difficult to fully support: one-past-the-end pointers.

Supporting one-past-the-end pointers in Archmage. Like many other previous models, Archmage represents memory as a set of *blocks*, which roughly correspond to consecutive regions of memory.

We call such a block-based representation of memory as *logical*, as these blocks do not have a physical manifestation. When a pointer is cast to an integer in Archmage, the block corresponding to a pointer is assigned a concrete physical address; such a representation is said to be *physical*.

Fig. 3.1 illustrates one of the reasons one-past-the-end pointers are challenging to support: their *ambiguity*. In Fig. 3.1, let us assume that blocks a and b have been allocated consecutively on the stack, and thus that the if-statement on line 4 evaluates to *true*.[1] Within the true-branch, we observe that the same integer value $i+4$ and j can be casted in *two ways*: (i) as a one-past-the-end pointer to block a or (ii) as a valid pointer to block b. The problem, then, occurs with the two writes on lines 6 and 8: the write on line 6 is a valid write to block a while the write on line 8 is a valid write to block b. However, the Quasi-Concrete model will fail to support this kind of pattern. This is because integers (that is, physical representations of a pointer) must be lifted to logical representations *at the time of the cast* (i.e., on line 5) in the Quasi-Concrete model. However, because $i + 4$ and j can be lifted in two ways, the Quasi-Concrete model must choose which representation to take before encountering the write on line 6—and will thus

```
char a[4], b[4];
i = (intptr_t) a;
j = (intptr_t) b;
if (i + 4 == j) {
  p = (char *) (i + 4);
  *(p - 1) = 42;
  q = (char *) j;
  *q = 37;
}
```

Figure 3.1: A code fragment illustrating integer-pointer casting with one-past-the-end pointers.

[1] We note that the fully logical memory model of CompCert actually does not support equality checks such as the one on line 4, because $i + 4$ is not a valid pointer to block a. Archmage, on the other hand, supports this check as $i + 4$ and j are both integers.

choose the 'valid' representation as a pointer to b on line 5, only to fail on the write on line 6. We observe that even if the Quasi-Concrete model chose to lift $i + 4$ as a one-past-the-end pointer to a, the write on line 8 would fail instead (as the cast on line 7 must return the same result, because $i + 4 = j$).

Archmage solves this problem by *lazily casting* integers to pointers only when required, and retaining the physical representation otherwise. That is, Archmage does not immediately lift $(i + 4)$ to a logical representation upon the cast on line 5; it instead simply postpones the cast by retaining $i + 4$ in p. Archmage then performs the cast to block a to perform the load—note that, while the load is performed on the casted logical pointer a, the value retained in p remains the integer $i + 4$.

Afterwards on lines 7 and 8, $q = p$ is an integer, and Archmage can simply re-cast q to block b on line 8 as required. In essence, this lazy treatment has the effect of allowing physical representations to 'choose' their corresponding block when required, allowing Archmage to correctly model such complex behavior related to one-past-the-end pointers.

A key idea of Archmage that makes such lazy castings possible is that even if a physical pointer p has two possible logical representations (e.g., a normal valid pointer for block b and a one-past-the-end pointer for block a as in Fig. 3.1), the behavior of an operation, such as memory accesses and pointer comparisons, is guaranteed to condense into a single result: that is, there does not exist any scenario in which interpreting p as either a or b both lead to valid cases with different results (Thoerem 1). It must be the case that either a or b result in an invalid operation, or that the two results agree. In a sense, this is what guarantees that Archmage does not introduce additional unwanted behavior in order to capture complex one-past-the-end coding patterns.

One interesting feature of Archmage exposed by this lazy casting of integers

is that *integers refine pointers* in Archmage, in the sense that integers may be treated as pointers without any adverse effects. The fact that integers refine pointers whose physical addresses coincide with the integers brings with it a variety of benefits, such as allowing for a wider range of optimizations to be supported in Archmage. Integers refining pointers also introduces some challenges as well: for example, operations such as pointer comparison must now consider scenarios such as when one operand is a pointer and the other is an integer. Such challenges will be discussed in more depth when we introduce the memory model in detail (§4).

3.2 CompCertCast: Reconciling CompCert with Archmage

Based on Archmage, this paper then presents CompCertCast, which is an integration of Archmage into CompCert. CompCertCast represents, to the best of our knowledge, the first verified compiler with support for a wide range of integer-pointer casting idioms.[2]

Integrating the memory model of CompCert to work with Archmage represents a significant engineering effort that required many detailed updates to existing CompCert infrastructure, such as extending the external call axioms, or adding additional simulation relations: details about these modifications may be found in §5. In addition to these modifications, CompCertCast also brings with it two new ideas: (*i*) an improvement of the "lower bound" on the assembly that CompCert generates, and (*ii*) a new optimization that mitigates the overhead from formally considering integer-pointer casts.

The "lower bound" improvement is an additional proof that fully concretizes

[2]CompCertS is also a verified compiler with support for integer pointer casts. However, CompCertS is incapable of supporting certain commonly used integer-pointer casting patterns (e.g., pointer hashing).

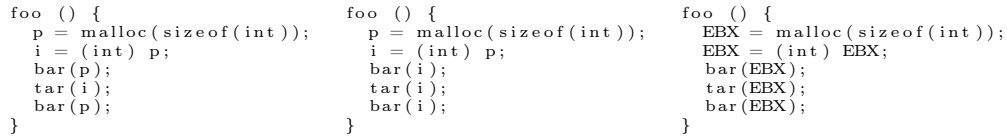

Figure 3.2: An example of decreasing register pressure via applying cast propagation.

the memory model used by CompCert-assembly, which is actually a fully logical memory model, into a fully physical model. That is, while assembly generated by original CompCert operates on a logical memory model, CompCertCast further guarantees that this assembly can operate in fully physical memory where all pointers are concretized as integers. This brings the assembly generated by CompCertCast a step closer to actual assembly that can be run on a processor.

Second, CompCertCast also introduces a new optimization, which we call *cast propagation*, in order to mitigate the performance overhead from formally considering integer-pointer casts. The intuition is that for a pointer p and its integer representation i, it is sound in Archmage to replace occurrences of p with i because integers refine pointers. In turn, this allows a limited form of copy propagation to occur by replacing p with i.

One important effect cast propagation has is reducing the *register pressure* of compiled code. CompCert performs register allocation on an intermediate representation in which logical pointers are retained, which creates a problem in which a *single* pointer with both logical and physical representations consumes *two* registers during register allocation: one for each representation. Fig. 3.2 illustrates such a scenario, where in the leftmost code snippet, where i is an integer-cast of p— p and i are allocated different registers because they have different values with overlapping lifetimes.

Applying cast propagation allows us to replace the occurences of p on lines 4

and 6 with i instead (as is in the middle of Fig. 3.2). Then, because the lifetime of p is up to the right-hand side of line 3, the register EBX can be reused to store i—which ultimately results in the code snippet using only one register, as illustrated in the rightmost of Fig. 3.2.

Cast propagation also exposes places where further optimizations, such as common subexpression elimination, may be applied. Details on the lower bound improvement, cast propagation, and the engineering required to reconcile CompCert with Archmage in general can be found in §5.

3.3 Archmage Logic

Finally, this paper also presents Archmage logic, a source-level separation-logic style proof system that captures the semantics of statements related to integer-pointer casts as inference rules, and allows users to *write* and *verify* specifications on such programs using these inference rules. Archmage logic completes our end-to-end verification chain: with Archmage logic, it is possible to perform source-level verification for programs with integer-pointer casts, which can then be compiled down into a verified binary with CompCertCast.

Archmage logic is designed with *usability* as a primary goal, such that using Archmage logic as a tool for verifying programs is as simple as possible. In particular, Archmage logic is designed to abstract much of the details about integer-pointer casts away, and instead provide a clean interface in which users can seamlessly transition between logical and physical representations of a pointer. To this end, Archmage logic provides users with three main predicates that capture the semantics of integer-pointer casts, where m represents block data (a, sz) for a block a and its size sz:

- $p_1 \approx^{\mathsf{m}} p_2$, indicating that two pointers (either physical or logical) p_1 and p_2 are equivalent,

- $p \mapsto_q^{\mathsf{m}} v$, indicating that a pointer p points to a location containing the value v,
- $\mathsf{live}_q^{\mathsf{m}}(p)$, indicating that a pointer p is at the beginning of a live (i.e., not freed) block m.

Note that the first predicate is *persistent* (*i.e.,* freely duplicable, seen as *knowledge*), whereas the others are not (seen as *ownership*). Moreover, the second and third predicates have fractional permission (or ownership) q, where $0 < q \leq 1$. Intuitively, operations such as writes to p, which may cause race conditions, may only occur when one can establish that $p \mapsto_1^{\mathsf{m}} v$ (i.e., when $q = 1$). In contrast, benign operations such as read may occur with $q < 1$. This model of fractional permissions is a standard idea that has been used in many separation logics; we refer the reader to [15, 16] for details.

The first predicate represents a core feature of Archmage logic: given two (either logical or physical) pointers p_1 and p_2, $p_1 \approx^{\mathsf{m}} p_2$ encodes that they are equal, or one is the physical address of the other. This relation gives a notion of *equivalence*, which allows one to freely substitute p_1 for p_2 (and vice versa) in the logic. To see how this substitution principle works in practice, consider the small code snippet depicted in Fig. 3.3. On line 3, the pointer-to-integer cast generates that $a \approx^{\mathsf{a}} i$. Then, on the precondition for the write to line 5, we will have that $a{+}1 \mapsto_1^{\mathsf{a}} \mathit{undef}$ (from line 2 upon allocation of a), which simply states that $a + 1$ points to an uninitialized value *undef*, that $a \approx^{\mathsf{a}} i$ (from line 3), and that $p = i{+}1$ (from line

```
foo () {
  char a[4], b[4];
  i = (intptr_t) a;
  p = (char *) (i + 1);
  *p = 42;
  j = (intptr_t) b;
  if (i + 4 == j)
    *(b - 3) = 37;
  return (p == a + 1)
}
```

Figure 3.3: A code fragment to illustrate the use of Archmage logic.

4 because the integer representation is retained). Then Archmage logic allows the following inferences:

- $a + 1 \approx^{\mathsf{a}} i + 1$ (adding offsets to the equivalence relation),
- $a + 1 \approx^{\mathsf{a}} p$ (since $i + 1 = p$),
- $p \mapsto_1^{\mathsf{a}} \mathit{undef}$ (replacing $a + 1$ with p in $a + 1 \mapsto_1^{\mathsf{a}} \mathit{undef}$).[3]

As line 5 can be proven to have write permissions to p, it can generate the postcondition $p \mapsto_1^{\mathsf{a}} 42$.

Lines 6-8 of Fig. 3.3 also illustrates why it is necessary to have the block data annotation m over $\approx^{\mathsf{m}}$. Let us consider a scenario *without* m, and assume that a and b are allocated consecutively on the stack. Then, in this scenario, we have that $b \approx j$ (from line 6) and that $i + 4 = j$ (reaching line 8). Staring from $b \approx j$, one can obtain: (*i*) $b - 3 \approx j - 3$ (offset subtraction), (*ii*) $b - 3 \approx p$ (since $j - 3 = i + 1 = p$), and (*iii*) $b - 3 \mapsto_1 42$ (replacing p with $b - 3$ in $p \mapsto_1 42$). Thus it becomes possible to access $b - 3$ at line 8, which is unsound in Archmage because $b - 3$ is an out-of-range logical pointer. Also note that according to the C standard, line 8 must be inaccessible triggering *undefined behavior* (otherwise, many optimizations become difficult to justify because become alias analyses impossible to perform). In contrast, adding the block data annotation as in Archmage prevents the third inference, as $b - 3 \approx^{\mathsf{b}} p$ (from the first and second inferences) and $p \mapsto_1^{\mathsf{a}} 42$ are defined over different block data (*i.e.,* a and b) and thus the substitution principle does not apply.

The third predicate $\mathsf{live}_q^{\mathsf{m}}(p)$ is used to encode that a pointer p is associated with a live block m, required for validating, *e.g.,* comparisons on p. Continuing with the example in Fig. 3.3, one can prove that $\mathsf{p} == \mathsf{a} + 1$ at line 9 evaluates

[3]Such replacements are only possible if the block in question a is identical; such details are formalized in §6.

to *true* in Archmage logic as follows. Starting from $\mathsf{live}^{\mathsf{a}}_1(a)$ and $\mathsf{a}.sz = 4$ (from line 2 upon allocation of a), $a \approx^{\mathsf{a}} i$ (from line 3), and $p = i + 1$ (from line 4), one can first obtain $\mathsf{live}^{\mathsf{a}}_{0.5}(a) * \mathsf{live}^{\mathsf{a}}_{0.5}(a)$ (by splitting the permission), from which $\mathsf{live}^{\mathsf{a}}_{0.5}((a + 1) - 1) * \mathsf{live}^{\mathsf{a}}_{0.5}(p - 1)$ (by applying the substitution principle for $a \approx^{\mathsf{a}} p - 1$) follows. This means that $a + 1$ and p resolve to the same live block a and offset 1. Since the offset 1 is in the weak valid range of a (*i.e.,* $0 \leq 1 \leq \mathsf{a}.sz$), one can prove that `foo` returns true.

The fact that we separate the liveness and read / writeability predicates, in tandem with the fractional permissions, allow Archmage logic to capture the permission model of CompCert (e.g., in CompCert, threads are allowed to compare pointers without knowing what their contents are).

Despite Archmage logic being designed to be a succinct and easy-to-use logic, it is nevertheless powerful enough to prove the majority of properties of interest for integer-pointer casting programs. In particular, Archmage logic does not require any modification to the source language in order to capture the semantics of integer-pointer casts, and can operate directly on the source program instead. As an example of the utility of Archmage logic, we present a correctness proof of an xor-linked-list implementation in §6.2. To the best of our knowledge, this proof is the first correctness proof of a xor-linked-list that operates on a memory model with logical blocks and integer-pointer casts, and thus allows for end-to-end verification of the xor-linked-list (in contrast, the proof given by [17] operates on a flat memory model and thus does not compose well with compiler optimizations).

Chapter 4

The Memory Model Archmage

This section presents a formal view of Archmage, a memory model developed with the goal of facilitating end-to-end verification for C programs containing integer-pointer casts. In this paper, we assume 64-bit integers (i.e., integers are 8 bytes in size).

4.1 The Definition of Archmage

Fig. 4.1 contains the definitions for the various constructs required for Archmage, and Fig. 4.2 contains some selected operational semantics for memory operations within Archmage. We will reference these two figures to explain Archmage: first starting with how memory and pointers are defined in *Archmage*, then explaining the operational semantics.

Definition of the Memory Model. In Archmage, memory is defined as a set of indexed logical *blocks*; the indices are called $\mathsf{BlockIDs}$ (modeled simply via the naturals), as captured by the partial function definition $\mathsf{Mem} \stackrel{\text{def}}{=} \mathsf{BlockID} \xrightarrow{fin}$

$$M \in \mathsf{Mem} \stackrel{\text{def}}{=} \mathsf{BlockID} \xrightarrow{fin} \mathsf{Block}$$
$$b \in \mathsf{BlockID} \stackrel{\text{def}}{=} \mathbb{N} \quad sz \in \mathsf{Size} \stackrel{\text{def}}{=} \mathbb{N} \quad v \in \mathsf{Val} \stackrel{\text{def}}{=} \mathsf{Int} \uplus \mathsf{LogicalPtr} \uplus \{undef\}$$
$$\mathsf{Block} \stackrel{\text{def}}{=} \{\ (live, pa, sz, c) \mid live \in \mathbb{B} \wedge pa \in \mathsf{Int} \uplus \{undef\} \wedge sz \in \mathsf{Size} \wedge c \in \mathsf{Val}^{sz}\ \}$$
$$p \in \mathsf{Pointer} \stackrel{\text{def}}{=} \mathsf{LogicalPtr} \uplus \mathsf{Int} \qquad (b, ofs) \in \mathsf{LogicalPtr} \stackrel{\text{def}}{=} \mathsf{BlockID} \times \mathsf{Int}$$

Figure 4.1: Type definitions for Archmage.

Block. Blocks in Archmage are what consist the actual memory layout, where a block is defined as a tuple of four elements: (*i*) a Boolean *live* that denotes whether the block is live or dead (a free block will be dead); (*ii*) an integer *pa* that denotes the physical address of this logical block, which may be undefined for blocks that have yet to receive a physical address; (*iii*) an integer *sz* that denotes the size of the block, and (*iv*) a list of values of length *sz*, that contains the actual contents of the block. *Values* in Archmage are assumed to be integers, logical pointers, or undefined values.

Pointers in Archmage are defined as logical pointers (LogicalPtr in Fig. 4.1) or physical pointers (integers). Logical pointers in Archmage are equal to pointers in the original CompCert. A logical pointer is a tuple of a blockID *b* (i.e., the block that the logical pointer points to) and an offset *ofs* that indicates the offset within the block. A physical pointer is simply an integer representing a physical address *pa*: we assume that physical pointers may access *any* logical block, given that *pa* coincides with the physical address assigned to that logical block.

Remark. *As we mentioned in §1.2.1, we simplified figures in this section for presentation. The actual Coq formalization, which extends the memory model of CompCert, does not have this simplification and has a faithful representation of the size of values, and the alignment constraints that come with it, instead.*

Operational Semantics of Memory Operations. Having understood memory representation in Archmage, we now describe how memory operations manipulate

memory through the semantics presented in Fig. 4.2.

A fresh allocation in Archmage takes the size sz as argument, then creates a new block where the physical address and the contents of the block are undefined (the first rule in Fig. 4.2). A free operation will first convert a pointer (either physical or logical) into a logical block (via toPtr defined in Fig. 4.2) and check if the block can be freed (captured by the premise $m(b) = (true, pa, n, c)$ in the second rule of Fig. 4.2), then proceed to update the memory with the information that the block has been freed (the conclusion of this rule). Similarly, for memory accesses, Archmage will convert a pointer into a logical block and perform the access according to the logical block.

Moving on to integer-pointer casting, casting in Archmage is performed in a similar manner to the Quasi-Concrete model [4]. A pointer-to-integer cast on a logical pointer (b, ofs) will either (i) assign a new physical address pa to the associated block if it does not have a physical address (as shown in the rule), or (ii) simply return pa of the associated block if the block already has a physical address. If there are no available physical address to allocate, Archmage triggers out-of-memory which is modeled as *no behavior* (NB, i.e., the program will do *nothing*) in Archmage. On a physical pointer, a cast will simply return the address $paddr$. These behaviors are formalized by the four rules for ptoi in Fig. 4.2, on the second and third rows.

On the other hand, integer-to-pointer casts are straightforward: an integer i is cast into a physical pointer i (the rule for itop in Fig. 4.2).

It is important to note that fresh allocations within Archmage result only in logical pointers, which are again *lazily* assigned physical addresses when required by a pointer-to-integer cast. We will later illustrate that preserving a logical-only representation of pointers for as long as possible also allows Archmage to support a wider range of optimizations (for comparison, consider the fact that CompCert

supports the full range of optimizations by virtue of having only logical pointers).

Integers Refine Pointers: Semantics of Binary Operations on Pointers. As briefly mentioned in §3, Archmage allows integers to refine pointers in order to achieve a variety of benefits (e.g., additional optimizations such as cast propagation). However, the fact that integers may refine pointers means that operations defined on pointers must now be defined on integers as well. Defintions for these operators can often naturally be extended by casting the integer to a pointer (e.g., for loads) or directly performing the operation on the integer (e.g., for pointer-offset addition), but an especially subtle case arises for binary operations (denoted as $\otimes$ in Fig. 4.2), in which operations may be supplied mixed operands (e.g., a logical pointer and an integer).

Archmage systematically extends the original semantics of CompCert $[\![\otimes]\!]_{\text{CompCert}}$ to handle such possibilities. We first observe that binary operations between pointers are limited to the case of subtraction and comparison. For a comparison operator $\otimes$, when one operand is a logical pointer p and the other is a non-null integer i, we consider two possible scenarios: lifting i to a logical pointer via toPtr, and concretizing p to a physical one via toInt. Archmage then takes the meet (i.e., intersection) of the result of the two scenarios, as defined via the meet $v_1 \mathbin{\overline{\wedge}} v_2$ in Fig. 4.2. Intuitively, this may be as that if either the concretization of p or lifting of i is undefined, then that scenario will return *undef* and Archmage will select the behavior of the other scenario by taking intersection. Otherwise, when the type of the operands are identical, Archmage applies the original CompCert semantics.

For subtraction, first note that CompCert uses separate operators (via overloading) for subtraction between pointer types (named psub) and subtraction between other types (named npsub) in the typed source language Clight, which are unified into a single operator sub when translated down to untyped inter-

mediate languages. However, Archmage does not perform this unification: to understand why, consider a subtraction $p - i$ between a logical pointer p and an integer i. $p - i$ has two possible interpretations in Archmage: (i) when i is an actual integer, upon which $p - i$ is an offset subtraction, or when (ii) when i is the physical representation of a pointer, upon which $p - i$ is pointer subtraction. The semantics of these two cases are different, and thus *Archmage* maintains the separation between psub and npsub to distinguish between these scenarios.

Then to define the semantics of subtraction, we give separate semantics for psub and npsub. both of which extend the original semantics of CompCert. $[\![\mathsf{psub}]\!]_M$ is defined in a similar way to comparison: it takes the intersection of the two possible scenarios, but does not consider the null pointer (physical address 0) as a special occasion. The rule in Fig. 4.2 does not explicitly invoke the meet operator: this is because, if the toPtr_M case does not result in `vundef` while computing $[\![\mathsf{sub}]\!]$, the toInt_M case is also guaranteed not to result in `vundef`. Because the meet operation is guaranteed to succeed (Theorem 1), this allows us to take the toInt_M case exclusively in the definition for simplicity. npsub covers the rest of the possible scenarios: the original semantics cover all cases except when both operands are (logical) pointers, in which case npsub produces *undef* in Archmage.

Here, an important part to note is that the meet operation $\overline{\wedge}$, as defined in Fig. 4.2, is *guaranteed* to succeed when combining the results of concretizing a pointer and lifting an integer, for all binary operations.

Theorem 1. *For any binary operator* $\otimes$ *and values* v_1, v_2*:* $[\![\otimes]\!]_M\,(v_1, v_2) \neq \mathit{NB}$

Handling Out-of-Memory. Archmage formally treats out-of-memory by modelling out-of-memory as *no behavior* (NB), following previous work such as the Quasi-Concrete model or CompCert-TSO. NB is a dual notion of *undefined behavior*

(UB), where a program will do *nothing* after triggering NB (versus doing anything after triggering UB). Because Archmage formally models out-of-memory, the guarantees provided by Archmage are sound even for programs that may trigger out-of-memory.

One drawback of modelling out-of-memory as no behavior is that optimizations that *reduce physical memory* are unsound in Archmage, as such optimizations may remove from the target an occurrence of out-of-memory that appears in the source. Archmage attempts to alleviate as much of this overhead as much as possible by assigning physical addresses to logical pointers as lazily as possible (i.e., only when a cast is met during execution). This allows Archmage to still perform memory-reducing optimizations (such as pure call elimination) provided that the optimization does not reduce consumption of the physical address space.

$$\frac{b \text{ is fresh} \quad \mathrm{blk} = (\mathit{true}, \mathit{undef}, sz, \mathit{undef}^{sz})}{(\mathsf{alloc}(sz), M) \to ((b,0), M[b \mapsto \mathrm{blk}])} \qquad \frac{\mathsf{toPtr}_M(p) = (b,0) \quad M(b) = (\mathit{true}, pa, sz, c)}{(\mathsf{free}(p), M) \to ((), M[b \mapsto (\mathit{false}, pa, sz, c)])}$$

$$\frac{\mathsf{toPtr}_M(p) = (b, \mathit{ofs}) \quad M(b) = (\mathit{true}, sz, c) \quad c' = c[\mathit{ofs} \mapsto v]}{(\mathsf{store}(p, M) \to ((), M[b \mapsto (\mathit{true}, sz, c')])}$$

$$\frac{\mathsf{toPtr}_M(p) = (b, \mathit{ofs}) \quad M(b) = (\mathit{true}, sz, c) \quad c[\mathit{ofs}] = v}{(\mathsf{load}(p), M) \to (v, M)}$$

$$\frac{M(b) = (\mathit{live}, \mathit{undef}, sz, c) \quad pa \in \mathsf{valid\_pa}(M, sz)}{(\mathsf{ptoi}((b, \mathit{ofs})), M) \to (pa + \mathit{ofs}, M[b \mapsto (\mathit{live}, pa, sz, c)])} \qquad \frac{M(b) = (\_, pa, \_, \_) \quad pa \neq \mathit{undef}}{(\mathsf{ptoi}((b, \mathit{ofs})), M) \to (pa + \mathit{ofs}, M)}$$

$$\frac{M(b) = (\_, \mathit{undef}, \_, \_) \quad \mathsf{valid\_pa}(M, sz) = \emptyset}{(\mathsf{ptoi}((b, \mathit{ofs})), M) \to \mathtt{NB}} \qquad \frac{pa \in \mathsf{Int}}{(\mathsf{ptoi}(pa), M) \to (pa, M)}$$

$$\frac{}{(\mathsf{itop}(i), M) \to (i, M)} \qquad \frac{}{(v_1 \otimes v_2, M) \to ([\![\otimes]\!]_M(v_1, v_2), M)}$$

$$\mathsf{range}(\mathrm{blk}) \stackrel{\text{def}}{=} \mathtt{if}\ \mathrm{blk}.\mathit{live} \wedge \mathrm{blk}.pa \neq \mathit{undef}\ \mathtt{then}\ [\mathrm{blk}.pa,\ \mathrm{blk}.pa + \mathrm{blk}.sz - 1]\ \mathtt{else}\ \emptyset$$

$$\mathsf{valid\_pa}(M, sz) \stackrel{\text{def}}{=} \{pa \mid sz > 0 \wedge [pa,\ pa + sz - 1] \subseteq ((0, \mathrm{INTMAX}) \setminus \textstyle\bigcup_{(b,\mathrm{blk}) \in M} \mathsf{range}(\mathrm{blk}))\}$$

$$\begin{aligned}\mathsf{toPtr}_M(v) \stackrel{\text{def}}{=}\ & \mathtt{match}\ v\ \mathtt{with} \\ & \mid \mathit{undef} \mid (b, \mathit{ofs}) \Rightarrow v \\ & \mid i \Rightarrow \mathtt{if}\ \exists (b, \mathrm{blk}) \in M,\ i \in \mathsf{range}(\mathrm{blk})\ \mathtt{then}\ (b, i - \mathrm{blk}.pa)\ \mathtt{else}\ \mathit{undef}\end{aligned}$$

$$\begin{aligned}\mathsf{toInt}_M(v) \stackrel{\text{def}}{=}\ & \mathtt{match}\ v\ \mathtt{with} \\ & \mid \mathit{undef} \mid i \Rightarrow v \\ & \mid (b, \mathit{ofs}) \Rightarrow \mathtt{if}\ M(b).pa \neq \mathit{undef}\ \mathtt{then}\ M(b).pa + \mathit{ofs}\ \mathtt{else}\ \mathit{undef}\end{aligned}$$

$$\begin{aligned}v_1 \mathbin{\bar{\wedge}} v_2 \stackrel{\text{def}}{=}\ & \mathtt{match}\ v_1, v_2\ \mathtt{with} \\ & \mid \_, \mathit{undef} \Rightarrow v_1 \\ & \mid \mathit{undef}, \_ \Rightarrow v_2 \\ & \mid \_, \_ \Rightarrow \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ v_1\ \mathtt{else}\ \mathtt{NB}\end{aligned}$$

$$\begin{aligned}\mathsf{lift}_M(f)(v_1, v_2) \stackrel{\text{def}}{=}\ & \mathtt{if} \quad (v_1, v_2) \in \mathsf{LogicalPtr} \times (\mathsf{Int} \setminus \{0\})\ \mathtt{then}\ f(v_1, \mathsf{toPtr}_M(v_2)) \mathbin{\bar{\wedge}} f(\mathsf{toInt}_M(v_1), v_2) \\ & \mathtt{elif}\ (v_1, v_2) \in (\mathsf{Int} \setminus \{0\}) \times \mathsf{LogicalPtr}\ \mathtt{then}\ f(\mathsf{toPtr}_M(v_1), v_2) \mathbin{\bar{\wedge}} f(v_1, \mathsf{toInt}_M(v_2)) \\ & \mathtt{else}\ f(v_1, v_2)\end{aligned}$$

$$\begin{aligned}[\![\mathsf{psub}]\!]_M(v_1, v_2) \stackrel{\text{def}}{=}\ & \mathtt{if} \quad (v_1, v_2) \in \mathsf{LogicalPtr} \times \mathsf{Int}\ \mathtt{then}\ [\![\mathsf{sub}]\!]_{\mathrm{CompCert}}(\mathsf{toInt}_M(v_1), v_2) \\ & \mathtt{elif}\ (v_1, v_2) \in \mathsf{Int} \times \mathsf{LogicalPtr}\ \mathtt{then}\ [\![\mathsf{sub}]\!]_{\mathrm{CompCert}}(v_1, \mathsf{toInt}_M(v_2)) \\ & \mathtt{else}\ [\![\mathsf{sub}]\!]_{\mathrm{CompCert}}(v_1, v_2)\end{aligned}$$

$$[\![\otimes]\!]_M(v_1, v_2) \stackrel{\text{def}}{=} \begin{cases} \mathsf{lift}_M([\![\otimes]\!]_{\mathrm{CompCert}})(v_1, v_2) & \text{if } \otimes \text{ is a comparison} \\ [\![\mathsf{psub}]\!]_M(v_1, v_2) & \text{if } \otimes = \mathsf{psub} \\ \mathtt{if}\ v_1, v_2 \in \mathsf{LogicalPtr}\ \mathtt{then}\ \mathit{undef}\ \mathtt{else}\ [\![\mathsf{sub}]\!]_{\mathrm{CompCert}}(v_1, v_2) & \text{if } \otimes = \mathsf{npsub} \\ [\![\otimes]\!]_{\mathrm{CompCert}}(v_1, v_2) & \text{otherwise} \end{cases}$$

Figure 4.2: Selected and simplified rules for the semantics of Archmage.

Chapter 5

CompCertCast: Reconciling CompCert with Archmage

Having formally defined Archmage, we now turn to task of reconciling Archmage with CompCert in order to create CompCertCast, an extension of CompCert that provides correctness of compilation and optimizations for code that contains integer-pointer casts. CompCert is the de-facto standard of a verified C compiler supporting many optimizations, allowing us to both (*i*) avoid having to re-establish the soundness of existing optimizations unrelated to integer-pointer casting, and (*ii*) provide guarantees on compiling integer-pointer casts in a practical framework used by a wide audience.

The main challenge in extending CompCert to support integer-pointer casts, is that we must replace the underlying memory model from the current logical model (which, as discussed in §4, only has very limited support for integer-pointer casting) to Archmage. Such a replacement of course brings with it a host of complications, as replacing a memory model has an effect on the semantics for a language, and thus, for example, existing soundness theorems must be

re-established if they are affected by this change. As discussed in §3, adding support for integer-pointer casts also has an effect on the performance of the final compiled result, as some optimizations become difficult to justify. On the other hand, supporting integer-pointer casts—and in particular, allowing integers to refine pointers, as in Archmage—also allows a new benefit, in that the 'lower-bound' of assembly generated by CompCertCast can now be configured to operate entirely over integers, and completely hide logical pointers (which is arguably closer to how machine code operates).

In this section, we provide a detailed view of the aforementioned challenges and benefits: §5.1 explains technical details related to updating CompCert, §5.2 details steps towards mitigating the performance overhead caused by formally considering integer-pointer casts, and §5.3 explains the improvement on generated assembly.

CompCertCast is available as a fully proved Coq implementation.

5.1 Modifying CompCert to Support Integer-Pointer Casts

As discussed in §1.2, CompCert defines compiler correctness through *behavioral refinement* between a source program (SRC) and its target program (TGT). CompCert establishes behavioral refinement through showing that a *forwards simulation* holds between the source SRC and the target TGT. However, for a forwards simulation to imply behavioral refinement, the target program TGT must be deterministic. This assumption is a major point of incompatibility between CompCert and Archmage: Archmage introduces nondeterminism via pointer-to-integer casts (when physical addresses are assigned to a block) as a block may be assigned any valid physical address.

```
Inductive val_intptr_M : Val → Val → ℙ :=
  | ...
  | val_intptr_ptr_int :
    toInt_M (b, ofs) = i → (i ≠ undef) → val_intptr_M (b, ofs) i
```

Figure 5.1: Part of the definition of the function $\mathsf{val\_intptr}_M$, for the case where the first argument is a pointer (b, ofs) and the second is an integer i. In this case, $\mathsf{val\_intptr}_M$ checks whether (b, ofs) has the integer representation i through toInt_M (defined in Fig. 4.2).

5.1.1 Mixed Simulations and Memory Relations

To address these challenges, we modify CompCert to show behavioral refinement by extending the concept of *mixed simulations* [18] instead of relying solely on forwards simulations. Intuitively, a mixed simulation holds if (i) a forwards simulation holds and the target is locally deterministic [18][1], or (ii) a *backwards* simulation holds, which is similar to the concept of a forwards simulation except that there must exist a corresponding step in SRC for each execution step in TGT.

The main contribution that CompCertCast provides in terms of proving refinement is the extension of the memory relations in CompCert to work with a concrete memory model as well. In original CompCert, this requirement is less of a problem because only values of the same type (aside from *undef*) may refine each other. However, in CompCertCast, *integers refine pointers*—and in particular, we would like to take advantage of this fact in order to apply optimizations such as cast propagation, as illustrated in §3.2. CompCertCast thus defines an additional memory relation, in addition to the three existing

[1] 'Locally deterministic' means that the state transition machine corresponding to the program is deterministic (i.e., has only one possible transition) at the current (i.e., local) state. Mixed simulations allow for a program to have a mix of deterministic and non-deterministic states: one uses forward simulations for the locally deterministic states, and backward simulations for the non-deterministic ones.

memory relations in CompCert (identity, extension, and injection), to capture the fact that pointers may be refined by their underlying physical addresses. $\mathsf{val\_intptr}_M$ from Fig. 5.1 defines the refinement relation between pointers and integers for a memory M, using the relation toInt_M, which checks whether a logical pointer (b, ofs) has an integer representation i (toInt_M was previously defined in Fig. 4.2).

Similar to extending the memory relation to allow integers to refine pointers, one must also establish a refinement relation for different *events* in the source and target. For example, in CompCertCast, a system call that takes as argument a logical pointer p in the source may instead take as argument an integer i in the target, provided that i is the physical representation of p.

We observe that constructing a refinement relation between events is not as a straightforward task as it seems, even without considering the concretization of memory. The intuitive way to construct the relation might be to reference the current state of memory when constructing the simulation. However, such a relation would result in a very fragile refinement because it is difficult to relate between different pointers in source and target. For example, consider a simple print statement, `print(p)` for a pointer p. p may have the logical representation, e.g., $(b, sz) = (3, 0)$ in the source: but it is possible that in the target, p is assigned a different representation (e.g., $(2, 0)$), perhaps due to an optimization that removes an unused allocation.

CompCert circumvents this problem by only allowing "public global pointers", to be exposed via an event. Public global pointers do not allow optimizations to be performed on them and are thus guaranteed to have fixed logical representations in both source and target, which makes establishing the event refinement simple: the logical pointers must match. CompCertCast extends this idea towards integer-pointer casts by creating a 'initial map' when a program start, that

eagerly concretizes all public global pointers prior to execution. Then, it is possible to determine the refinement relation for events that expose a pointer in the source and an integer in the target by consulting the initial map.

Vertical composition of behavioral refinement that considers the aforementioned pointer-integer refinement can be then achieved by additionally requiring that the target-side map $\mathsf{init}_{\mathsf{TGT}}$ extends the source-side map $\mathsf{init}_{\mathsf{SRC}}$, *i.e.,* if the following equation is satisfied:

$$\mathrm{Beh}(\mathsf{TGT}) \leq_{\mathsf{init}_{\mathsf{TGT}}} \mathrm{Beh}(\mathsf{SRC}) \wedge \mathsf{init}_{\mathsf{SRC}} \leq \mathsf{init}_{\mathsf{TGT}}$$

It actually suffices that $\mathsf{init}_{\mathsf{SRC}} = \mathsf{init}_{\mathsf{TGT}}$ instead of $\mathsf{init}_{\mathsf{SRC}} \leq \mathsf{init}_{\mathsf{TGT}}$, but we just give a more general condition for vertical composition.

We observe that the eager concretization of public global pointers does not conflict with the rest of Archmage, which otherwise casts pointers to integers lazily. In particular, eagerly concretizing public global pointers has no detrimental effect on the performance of generated code, as optimizations cannot be applied to these pointers anyways.

One additional condition to note is that, optimizations that reduce the consumption of physical memory are unsound in CompCertCast as discussed in §3. Thus CompCertCast adds the additional condition that allocated physical memory in the target must extend the allocated physical memory in the source.

In addition to the developments related to the memory relation, CompCertCast also extends the mixed simulation defined in [18] to support out-of-memory as well. Given a TGT trace for which out-of-memory is triggered, there should exist a corresponding SRC trace such that the two traces match up until the point OOM takes place in TGT. Because Archmage interprets out-of-memory as *no behavior*, behavioral refinement is established if (*i*) TGT triggers OOM somewhere in the trace, and (*ii*) there exists a simulation between SRC and

TGT upto the point where OOM is triggered in TGT, exactly as captured by the aforementioned condition.

The idea of mixed simulation (which also considers out-of-memory) is formalized in Eqn. (5.1): When interpreting both SRC and TGT as state transition systems, as presented in §1.2, we say that a mixed simulation considering out-of-memory holds if the condition in Eqn. (5.1) is true.

$$
\begin{aligned}
\forall tr, st'_{\mathsf{SRC}},\ st_{\mathsf{SRC}} \overset{tr}{\hookrightarrow} st'_{\mathsf{SRC}} \implies & \exists st'_{\mathsf{TGT}}, tr', (st_{\mathsf{TGT}} \overset{\tau}{\hookrightarrow}^{*} \overset{tr'}{\hookrightarrow} \overset{\tau}{\hookrightarrow}^{*} st'_{\mathsf{TGT}} \wedge (st'_{\mathsf{SRC}}, st'_{\mathsf{TGT}}) \in R \\
& \qquad \wedge tr =_{\mathsf{init}_{\mathsf{TGT}}} tr'); OR \\
\forall tr, st'_{\mathsf{TGT}}, st_{\mathsf{TGT}} \overset{tr}{\hookrightarrow} st'_{\mathsf{TGT}} \implies & \exists st'_{\mathsf{SRC}}, tr', (st_{\mathsf{SRC}} \overset{\tau}{\hookrightarrow}^{*} \overset{tr'}{\hookrightarrow} \overset{\tau}{\hookrightarrow}^{*} st'_{\mathsf{SRC}} \wedge (st'_{\mathsf{SRC}}, st'_{\mathsf{TGT}}) \in R \\
& \qquad \wedge tr =_{\mathsf{init}_{\mathsf{TGT}}} tr') \vee \\
& \exists st'_{\mathsf{SRC}}, tr', (st_{\mathsf{SRC}} \overset{tr'}{\hookrightarrow} st'_{\mathsf{SRC}} \wedge \mathsf{oom}(tr) \\
& \qquad \wedge \mathsf{prefix}_{\mathsf{init}_{\mathsf{TGT}}}(tr, tr'))
\end{aligned}
\tag{5.1}
$$

In Eqn. (5.1), the top implication encodes the forwards simulation from Eqn. (1.1), with the additional requirement that transitions that occur in the target are fully deterministic. The bottom implication encodes the backward simulation, where one attempts to map transitions in the target to those in the source (instead of the other way around).

We observe that Eqn. (5.1) now matches a *trace of events* tr instead of a single event ev as in Eqn. (1.1): this is because a single transition may now trigger multiple events due to out-of-memory (i.e., a transition such as an external function call may trigger both its original event ev and an out-of-memory event at once), which Eqn. (5.1) must also support. Since source and target programs can produce different events, we modified the mixed simulation to ensure that events are equal with respect to the target program's 'initial map' (encoded as $=_{\mathsf{init}_{\mathsf{TGT}}}$ in Eqn. (5.1)). The out-of-memory condition appears in the fifth and sixth lines of Eqn. (5.1): given a TGT trace tr for which out-of-memory is triggered ($\mathsf{oom}(tr)$), this condition asks for a corresponding SRC trace tr' such that tr and tr' match up with respect to 'initial map' of target program

($\mathsf{init}_{\mathsf{TGT}}$) until the point OOM takes place in tr (expressed by the predicate $\mathsf{prefix}_{\mathsf{init}_{\mathsf{TGT}}}(tr, tr')$ in Eqn. (5.1)).

5.1.2 External Call Axioms

Similar to how the memory and event relations must be updated to support the addition of integer-pointer casts, the *external call axioms* of CompCert must also be updated to be sound under integer-pointer casts and the fact that integers refine pointers. Specifically, there are four main changes to the external call axioms:

- The axiom that external calls may only trigger one event has been removed: external calls may also trigger an additional out-of-memory after triggering whatever event they originally trigger.
- External call axioms for existing memory relations have been updated to work with backwards simulations.
- A new axiom stating that external calls may not 'tamper' with the memory map—e.g., an external call may not suddenly update the physical address of an already concretized pointer—has been added.
- A new axiom to let new memory relations (concrete_extends) work with backwards simulations has also been added.

An important part to note about the modifications to existing axioms (the first two changes) is that they are *relaxed* compared to the original external call axioms of CompCert, in the sense that if the original axioms hold then our modified axioms hold as well. Proofs that rely on the original external call axioms will thus still hold with the modified axioms, meaning that existing proofs (e.g., for optimizations) that make use of these axioms will still hold,

allowing us to reuse many proofs. Note that even under these additional axioms, CompCertCast maintains the original guarantee of CompCert for behavioral refinement under separate compilation.

5.1.3 Other Minor Modifications to the CompCert Infrastructure

In addition to the major changes to the memory relations and external call axioms, replacing the memory model of CompCert with Archmage to support integer-pointer casts also requires a host of smaller modifications to the existing CompCert infrastructure.

Supporting the Modified Semantics of Operations. Because the original source semantics used in CompCert only supports a narrow range of operations performed on values resulting from pointer-to-integer casts, we must extend the source semantics to support the full range of such operations in order to be able to use CompCert as a tool for end-to-end verification of programs with integer-pointer casts. Following this requirement, we have extended the source semantics of CompCert with the semantics of operations on values resulting pointer-to-integer casts as illustrated in §4.

Fixing Optimization Proofs. Finally, utilizing Archmage as the memory model of CompCert requires some fixes to the existing proofs of soundness for optimization in CompCert. Most heavily affected are optimizations that must deal with external calls (which include integer-pointer casts): as previously explained, external calls may be nondeterministic in Archmage and thus we revise the existing forwards-simulation based proofs in CompCert to use mixed simulations instead. In addition, because we have modified the semantics of some operators—such as the introduction of `psub`, or the semantics of pointer comparison—proofs for optimizations that deal with such modified operators must be fixed as well.

5.2 Identifying and Alleviating Performance Overhead

Although we have fixed proofs of soundness of optimizations in CompCert to work with Archmage in the previous section, supporting a formal model of integer-pointer casting and pointer arithmetic still incurs a performance overhead. There are various reasons for this performance overhead: most significantly, the additional complexity induced by integer-pointer casts renders CompCert *unable to recognize* certain patterns in which optimizations may be still applied in a sound manner. In this section, we identify and alleviate such performance bottlenecks in detail.

5.2.1 Cast Propagation: Replacing Uses of Pointers with Integers

One pattern in which a naive implementation of CompCertCast would miss optimization opportunities is *cast propagation*, as illustrated in §3.2. As discussed, the fact that integers refine pointers in Archmage allows us to replace all usages of a pointer p with its integer representation i—CompCertCast implements an additional pass that identifies such scenarios to apply cast propagation as much as possible.

In §3.2, we have already discussed how cast propagation, while seemingly simple, is essential in reducing the register pressure of the final compiled program. Here, we illustrate how cast propagation also plays a pivotal role in allowing CompCertCast to *identify further chances* for optimization. Fig. 5.2 gives an example of this phenomenon, where applying cast propagation opens up an additional chance to perform common subexpression elimination (CSE).

In Fig. 5.2, the leftmost code snippet shows a function where, the argument pointer p has been casted into a physical representation i, which is then re-cast to a pointer on line 4 for comparison with q. Here, one can see that without cast

```
int foo(long *p, long *q){
  int i = (int) p;
  int c1 = i < q;
  int c2 = p < q;
  return c1 + c2;
}
```

```
int foo(long *p, long *q){
  int i = (int) p;
  int c1 = i < q;
  int c2 = i < q;
  return c1 + c2;
}
```

```
int foo(long *p, long *q){
  int i = (int) p;
  int c1 = i < q;
  int c2 = c1;
  return c1 + c2;
}
```

Figure 5.2: An example application of common subexpression elimination, which may only take place after replacing `p` with `i` on line 4 through cast propagation.

```
foo(void *p){
  l1 = p;
  i = (int) p;
  l2 = p;
  l3 = malloc(8);

  memcpy(&l3, &l1, 4);
  memcpy(&l3+4, &l2+4, 4);
  // l3 = p
}
```

```
foo (void *p) {
  l1 = p;
  i = (int) p;
  l2 = i;
  l3 = malloc(8);

  memcpy(&l3, &l1, 4);
  memcpy(&l3+4, &l2+4, 4);
  // l3 = ?
}
```

Figure 5.3: An example illustrating the need to carefully define the semantics of load. The right code snippet is the result of applying cast propagation on the left code snippet.

propagation, CSE cannot be applied as there are no common subexpressions—however, applying cast propagation yields the second code snippet in Fig. 5.2, where lines 4 and 5 share the subexpression $i < q$. This then gives us an opportunity to apply CSE, ultimately yielding the final code snippet in Fig. 5.2.

One challenge that arises from applying cast propagation is that we must carefully define the semantics of load operations. Figure 5.3 gives an example of where load becomes problematic in the presence of cast propagation: in the left snippet, it is easy to infer that *l3* contains p after executing the two `memcpy`s on line 7 and 8. However, this inference becomes nontrivial in the right snippet, which is the result of applying cast propagation to the left: applying cast propagation sets *l2* to i, and thus *l3* now contains a mix of p and i. Then because p and i are of different type, a naive load of *l3* will yield *undef*—which makes cast propagation unsound.

CompCertCast solves this problem by simply treating pointers as integers

when performing the load for mixed values, which unifies the the mixed pointer and integer types in, e.g, *l3*, to just integers. Observe that we are *guaranteed* to be able to consider integer representations of pointers in these mixed scenarios, because the fact that cast propagation has occured on p implies that p has already been cast—and thus has a valid integer representation. Thus it is sound to use the integer representation of p instead when performing the load.

We argue that any model that supports integer-pointer casting will want to leverage the information that i is a physical representation of p in some way during their backend optimizations, and the fact that integers refine pointers combined with cast propagation gives Archmage a simple but highly elegant way to do so. The same cannot be said for, e.g., the Quasi-Concrete model [4], in which integers do not refine pointers and thus said optimizations are harder (or even impossible) to apply.

We note that cast propagation requires copy propagation and static single assignment (SSA) [19] to be fully effective, but original CompCert did not perform copy propagation to the degree which we were expected nor did it implement SSA. We thus utilized the SSA transformation pass from CompCert-SSA [20] and implemented a new copy propagation algorithm that relies on SSA, which is observed to be more efficient than that of original CompCert.

5.2.2 Flagging Stack Casts to Enable Stack-Local Optimizations

Another pattern in which a naive CompCertCast combination would fail to apply optimizations are some optimizations that are performed on instructions operating on *stack-local* variables. We take as example again common subexpression elimination.

Consider Fig. 5.4, which depicts a simple function `foo` which takes as argument a pointer p, reads from a stack-local array `stk`, write to p, then reads again from `stk`. According to C semantics, it is sound to replace line 6 with `int j = i` via applying CSE, regardless of the value that is passed through p. This is because p cannot be a pointer to the block that corresponds to the stack of `foo`, as a caller of `foo` is guaranteed not to have access to the stack pointer of `foo`.

```
int foo (void *p) {
  int stk[42];

  int i = stk[3];
  *p = 42;
  int j = stk[3];
  return i + j;
}
```

Figure 5.4: A small program that takes as argument a pointer `p` and writes to `p`.

However, recall that in Archmage, *physical representations* of pointers—that is, integers—have the ability to point to *any* logical block as long as the physical representation has a corresponding entry in Mem from Fig. 4.1, meaning that the write on line 5 could possibly write to the stack. Thus this optimization becomes harder to justify in Archmage—because the value of p is unknown, one must have a guarantee that the stack pointer of `foo` does not have a physical representation in Mem in order to apply the transformation in a sound manner.

To alleviate the aforementioned limitation, we implement an extra flag[2] that tracks whether a stack address has been cast to a physical address or supplied as an argument to an external call, as going through a cast is the *only* way a logical representation for the stack address to gain a physical representation (external calls are added because they may contain pointer-to-integer casts). Adding this flag allows us to apply CSE on the aforementioned pattern because it is now guaranteed that a physical pointer p will be unable to write to the

[2]CompCert already records various attributes of the stack; we simply add a Boolean flag to this data structure.

stack (since the stack has no physical representation).

We observe that despite implementing this flag, Archmage still is unable to justify such applications of CSE if there does exist a physical representation of the stack in Mem; e.g., when there is a pointer-to-integer cast of a stack address. This limitation may actually be alleviated if the stack is *split*—that is, different (address taken) variables are assumed to inhabit different logical blocks, and a stack is interpreted as the union of all such blocks—which would allow us to apply optimizations on sub-blocks that have not been cast, even if the address of some different stack variable has been captured. However, the current implementation of CompCert treats the whole stack of a function as a single block in RTL, the intermediate representation for main optimizations. We plan to enhance RTL to allow each function to have multiple stack blocks as in mainstream compilers such as LLVM.

5.3 The Lower Bound Improvement: Generating CompCert-Asm with Fully Physical Pointers

As discussed, CompCertCast allows us to achieve an improvement on the *lower bound* of generated code. This improvement is in the sense that CompCert-Asm (which we will from now on refer to as simply assembly) generated by CompCertCast may only contain physical pointers in memory and registers, as opposed to original CompCert, in which memory and registers may also contain logical pointers. Because memory and registers are the only locations where data can be stored at the assembly level, the lower bound improvement represents a *full concretization* of logical memory at the last step of the compilation chain.

The key idea in guaranteeing the physical lower bound is to extend the semantics of generated assembly, such that all logical pointers are concretized and replaced with their integer representations before each step of the assembly

semantics. For example, an allocation statement $x = \mathsf{alloc}(8)$ will (i) generate a new logical pointer p for $\mathsf{alloc}(8)$, (ii) *concretize p to i before the store to x*, which is the added step, then (iii) store i to the register mapped to x. The machinery developed previously in §5.1—the extended memory relations and event refinements, accounting for the fact that integers refine pointers—guarantee the soundness of such a semantics. Inserting the concretization step after each step of the assembly semantics allows us to correctly deal with external calls as well, which may insert logical pointers into memory and registers—the concretization step ensures that the changes imposed by external calls are all concretized, without having to impose additional axioms for external calls.[3]

We observe that the lower bound improvement brings the final assembly closer to an actual bare-metal model: the results of, e.g., allocations, can now all be treated as integers, and operations on pointers now happen all on their integer representations, just like real machine code.

5.4 Implementation

In this section, we give a brief discussion of the actual implementation of CompCertCast in Coq. Our implementation builds on top of CompCert version 3.9, while preserving the structure and optimizations of original CompCert as much as possible. Specifically, CompCertCast supports all compilation passes of original CompCert except for the front-end passes from C to Clight, while adding new optimization passes (shown in dotted boxes in Fig. 5.5).

In the optimization chain (the upper dotted box), SSAgen and De-SSA are an application of static-single assignment from CompCert-SSA [20]. Copy propagation is standard copy propagation; however, as CompCert did not

[3]Sometimes it may be the case that an external call inserts *malformed* logical pointers (e.g., a pointer to a block that has never been allocated). CompCertCast treats such scenarios as ill-formed, and ignores them.

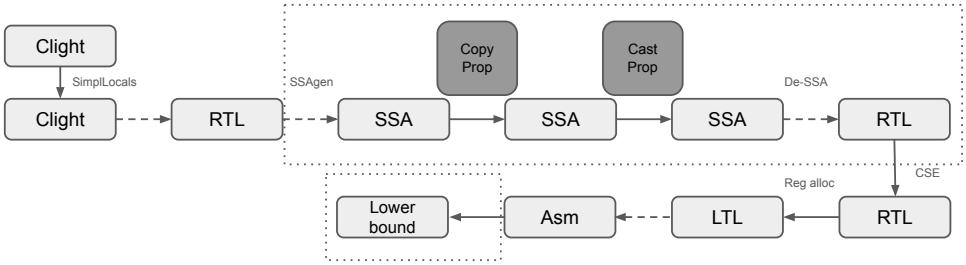

Figure 5.5: The additional compilation passes applied by CompCertCast, indicated by the dotted box. Our new passes copy and cast propagation, and the lower bound improvement are highlighted.

perform copy propagation optimally as discussed in §5.2, we implemented a new, more efficient version from scratch. Implementing these three additional optimizations were prerequisites to implementing cast propagation as described in §5.2; the final application of CSE is a reapplication of the original CSE pass to take advantage of cast propagation.

The lower bound improvement does not perform any changes to code, and instead simply extends the semantics of assembly as discussed in §5.3.

In terms of code size in Coq, replacing the memory model of CompCert with Archmage—that is, CompCertCast without the additional optimizations and lower bound guarantee—constitutes around a 24% increase in code. The implementation and proof of cast propagation took around an additional 6600 LoC, and the lower bound improvement took around an additional 3000 LoC.

We tested our implementation on existing CompCert benchmarks (which do not contain integer-pointer casts) and confirmed that for these benchmarks, CompCertCast emitted assembly identical to original CompCert except for a single function (`render_ray` from `render.c`). `render.c` is a case where CompCertCast cannot apply CSE because the stack has a physical representation, as

discussed in §5.2. We also tested that cast propagation was operating as intended on several small, hand-crafted programs. Interested readers may consult our Coq implementation [21].

Chapter 6

Archmage Logic

In this section, we present Archmage logic, which is a top-level proof system that captures the semantics of statements related to integer-pointer casts as inference rules, and allows users to *write* and *verify* specifications on such programs using these inference rules. In essence, the core of Archmage logic is a small, succinct set of rules that that capture the behavior of statements related to integer-pointer casts.

As the source language of Archmage, we port a version of Clight (the source language of CompCert), where gotos are removed, to interaction trees [22] to obtain Clight$^+$. Clight$^+$ is a suitable source language for end-to-end verification as we provide a refinement proof from Clight$^+$ to Clight. In practice, we implement Clight$^+$ and Archmage logic on top of the CCR verification framework [23], to obtain a verification interface for programs containing integer-pointer casts that may further be compiled by CompCert.

Although Archmage logic is succinct and easy to use, it is nevertheless powerful enough to prove the majority of properties of interest for integer-pointer

casting programs, by virtue of Archmage itself being designed as a memory model with end-to-end verification in mind. In particular, after introducing the rules of Archmage, we will show that Archmage logic can be used to express and prove the correctness of an XOR-based linked list and simple pointer hardening implementation. Existing verification frameworks are either incapable of supporting the complex integer-pointer casting patterns used in the xor-list [10], or are capable of verifying a source-level implementation, but fail to provide end-to-end guarantees throughout the compilation chain because the underlying program logic is inconsistent with compiler optimizations [17].

6.1 The Predicates and Rules of Archmage Logic

Archmage logic draws Iris-style separation logic built upon *resource algebras* (as introduced in [24]) in order to track the ownership of pointers. It thus follows that the predicates (*i.e.,* pre/postconditions) of Archmage logic are also written in the language of resources.

However, having to track and understand the semantics of such resources directly is complex, and stands in contrast with our end goal of usability. We thus apply an additional layer of abstraction, to obtain *user-level predicates* that hide underlying resources and instead expose an intuitive interface (*i.e.,* a set of properties about the user-level predicates). These user-level predicates are defined in 'Predicates and Relations' of Fig. 6.1, the former three of which encapsulate the underlying resources related to pointers. The user-level properties of these predicates are given in 'Selected Rules for Predicates and Relations' of Fig. 6.1. We will first explain these user-level predicates, then illustrate how the rules of Archmage logic capture the behavior of statements related to integer-pointer casts with these predicates.

From a user perspective, it suffices to carry the intuition that there are three

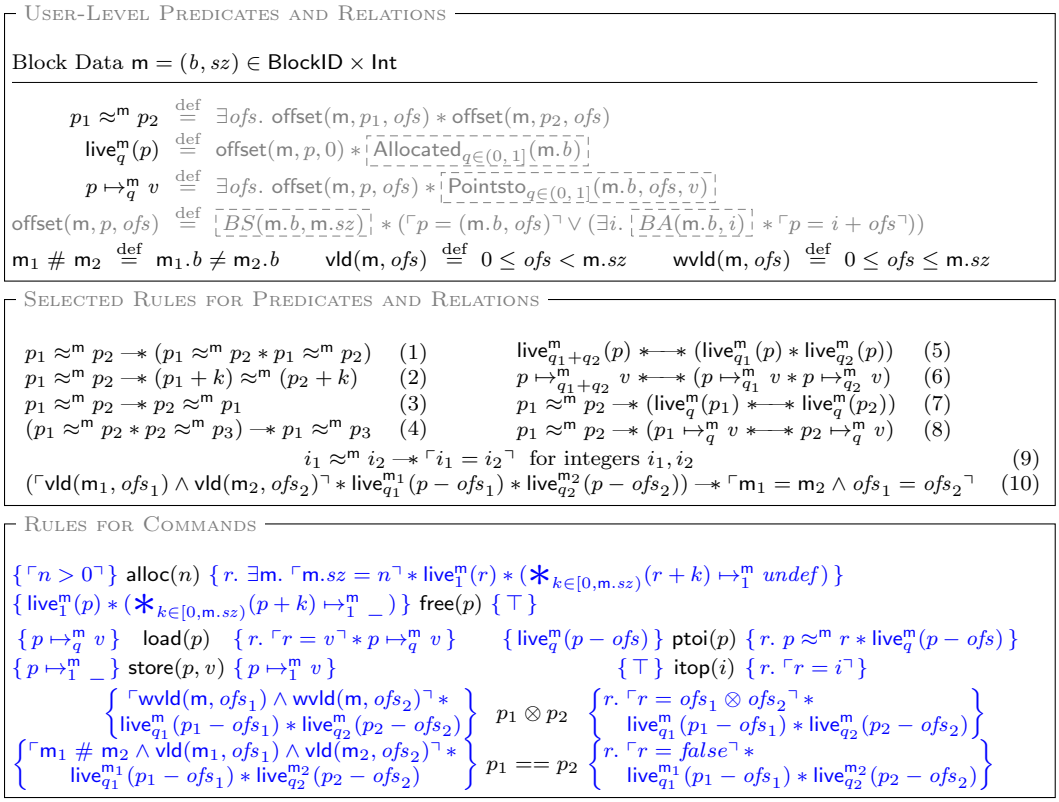

Figure 6.1: User-level predicates, command rules, and selected rules for predicates in Archmage logic.

main predicates related to pointers in Archmage logic. Given a pointer p with block data m:

- A predicate $p \approx^{\mathsf{m}} p'$ tracking *casting*: whether $p = p'$ or one is the physical address of the other.
- A predicate $\mathsf{live}^{\mathsf{m}}_q(p)$ tracking *liveness*: whether p is at the head of a live (*i.e.,* unfreed) block m,
- A predicate $p \mapsto^{\mathsf{m}}_q v$ tracking *accessibility*: whether p points to the value v with permission q.

As these are resource predicates in separation logic, they are created when a statement is executed and may be deleted by executing another statement. For example, one should not be able to derive that a fresh pointer p is live except from the result of an alloc statement. In addition, the latter two predicates must be *non-duplicable*. For example, having two copies of $p \mapsto^{\mathsf{m}}_1 v$ would allow simultaneous writes on p, possibly resulting in race conditions. However, at the same time, there must also exist a mechanism for dividing this predicate amongst multiple actors: while race conditions due to simultaneous writes must be blocked, simultaneous reads should be allowed. The accessibility predicate thus also carries a *fractional permission* $q \in (0, 1]$, where q may be divided amongst the actors that require access to p, and the *degree of accessibility* is determined by how much of q an actor possesses. For example, writes are only allowed with the precondition $p \mapsto^{\mathsf{m}}_1 v$: *i.e.,* when $q = 1$, or when the writer possesses the entirety of the permission. Also, since freed blocks should not be accessible, we delete the accessibility predicate with the entire permission (*i.e.,* $q = 1$) when executing a free statement. Similarly, we use fractional permission for the liveness predicate: while permission for liveness can be freely split as needed, the entire permission should be collected and deleted at deallocation.

In addition to these three resource predicates, there are three additional user-level pure (*i.e.,* without involving resources) predicates which encapsulate commonly used conditions: (*i*) $\mathsf{m}_1\#\mathsf{m}_2$, stating that the block data m_1 and m_2 are *different* blocks, (*ii*) $\mathsf{vld}(\mathsf{m}, \mathit{ofs})$, checking that *ofs* is a valid with respect to block data m (*i.e.,* that a pointer $p = (\mathsf{m}.b, \mathit{ofs})$ is an interior pointer), and (*iii*) $\mathsf{wvld}(\mathsf{m}, \mathit{ofs})$, encoding the same idea for weak validity (*i.e.,* including one-past-the-end pointers).

The exact definitions of these six predicates are given in the section 'User-Level Predicates and Relations' of Fig. 6.1. Among them, the definitions of the three resource predicates are given in grey color since they do not need to be visible to users. Note that these definitions involve four underlying resources (whose exact definitions can be found in our Coq development [21]): $BS(\mathsf{m}.b, \mathsf{m}.sz)$ and $BA(\mathsf{m}.b, i)$, persistent (*i.e.,* duplicable) resources capturing the size and physical address of a block; and $\mathsf{Allocated}_{q\in(0,\,1]}(\mathsf{m}.b)$ and $\mathsf{Pointsto}_{q\in(0,\,1]}(\mathsf{m}.b, \mathit{ofs}, v)$, fractional resources capturing the liveness and accessibility of a block.

Instead of providing the definitions, we provide abstract properties about the resource predicates that users will find useful when constructing proofs. A number of selected rules are listed in the section 'Rules for Predicates and Relations' of Fig. 6.1, where $\ulcorner - \urcorner$ is the lifting of a pure predicate into a resource predicate.

- (1): Casting predicates are duplicable.
- (2): One may add fixed offsets k to casting predicates to get predicates about the shifted location.
- (3)-(4): Casting predicates are symmetric and transitive.
- (5)-(6): Fractional permissions may be split into smaller values, or merged back into their sums.

- (7)-(8): (Substitution Principle) If one knows that $p_1 \approx^{\mathsf{m}} p_2$, then one may swap in p_2 for p_1.
- (9): The casting predicate coincides with equality on integer values.
- (10): The same pointer p cannot point to different live blocks with valid offsets.

Having understood the user-level predicates, we now proceed to describing how the rules of Archmage logic capture the behavior of commands containing pointers: these rules are listed in the box 'Rules for Commands' of Fig. 6.1, where pre-postconditions are highlighted in blue. We first observe the rule for alloc: this rule essentially states that performing a new allocation $p = \mathsf{alloc}(sz)$ with size $sz > 0$ *creates* two new resource predicates about the resulting pointer r: (i) $\mathsf{live}_1^{\mathsf{m}}(r)$, *i.e.,* that r is a live pointer pointing to the head of the newly allocated block m, and (ii) $(\divideontimes_{k\in[0,\mathsf{m}.sz)})(r+k) \mapsto_1^{\mathsf{m}} \mathit{undef})$, *i.e.,* that r has full write permissions to the whole block m containing uninitialized values *undef*. We observe that alloc is the *only* statement that can create the liveness and accessibility predicates: all other rules cannot generate these two predicates.

$\mathsf{free}(p)$, in turn, *consumes* both of the predicates generated by alloc: the rule requires the entire (*i.e.,* $q = 1$) liveness and accessibility predicates in the precondition, and consumes them both removing them from the postcondition. Because these two predicates are *non-duplicable*, and alloc is the only statement that can create these resources; consuming them via free guarantees that subsequent statements will be unable to doubly free p or read from / write to a freed pointer.

$\mathsf{ptoi}(p)$ is the final rule that can create or consume resource predicates: it requires a live pointer p in the precondition, and generates a new casting relation $p \approx^{\mathsf{m}} r$ in the postcondition, while also preserving the precondition

$\mathsf{live}^{\mathsf{m}}_{q}(p - ofs)$. In essence, $\mathsf{ptoi}(p)$ simply *adds* the knowledge (*i.e.,* persistent resource) that the pointer p now has a physical representation r.

The rest of the rules now merely check if the required resources are in place: for example, load requires that we have the fractional predicate $p \mapsto^{\mathsf{m}}_{q} v$ with $q > 0$ to read v from p, while store requires that we have the full accessibility predicate $p \mapsto^{\mathsf{m}}_{1} \_$ to write to p (and updates the predicate with the written value). The rule for $r = p_1 \otimes p_2$ captures pointer operations (*i.e.,* equality, comparison, and subtraction) for which p_1 and p_2 are pointers to the same block: the precondition requires that p_1 and p_2 are both live, in which case they should both be weakly valid, and the result of the operation may be computed from the offsets. The special rule for pointer quality $p_1 == p_2$ applies to cases where p_1 and p_2 point to different *live* blocks with *valid* offsets, for which the result is always *false*.

6.2 Case Study 1: Proving Correctness of a Xor-Based Linked List with Archmage Logic

To illustrate the power of Archmage logic, in this section we prove the correctness of a xor-based linked list implementation using Archmage logic. Xor-based linked lists (xor-lists) are space-efficient implementations of doubly linked lists: whereas an ordinary linked list requires two address entries for each node (previous and next node addresses prv and nxt), an xor-list requires only *one* address entry: one that stores the bitwise-xor of prv and nxt. (denoted as xor prv nxt). Traversal in the xor-list then proceeds by xor-ing this stored address with the address of a previous node—e.g., a heads-to-tail traversal will compute xor prv (xor prv nxt), where xor prv prv cancels out to yield nxt (the traversal function must provide prv as an argument).

Xor-lists are a prime example of an implementation that involves subtle

```
struct node {
  long item;  // value in node
  long link;  // xor prev next
}
long delete_hd(node** hdH, node** tlH) {
  long item = 0; // empty list
  node* hd_old = *hdH;

  if (hd_old != NULL) { // non-empty list
    item = hd_old->item;
    node *hd_new = (node*) hd_old->link;
    *hdH = hd_new;
    if (hd_new == NULL) {
      *tlH = NULL;
    } else {
      intptr_t link = hd_new->link;
      hd_new->link = link ^ (intptr_t)hd_old;
    }
    free(hd_old);
  }
  return item;
}
long delete_tl(node** hdH, node** tlH)
...
```

Figure 6.2: Snippets of an xor-list implementation, showing the code for struct `node` and the function `delete_hd`.

reasoning about pointer-integer casting, because (*i*) the pointers prv and nxt must be cast to integers to perform bitwise-xor, and (*ii*) this result must be reinterpreted as a pointer to read from or modify the list. Existing approaches to source-level verification of integer-pointer casting programs (that aim to also be consistent with compiler optimizations) are thus unable to verify the xor-list: for example VIP [10] fails on the xor-list because they cannot resolve which block xor prv (xor prv nxt) should point to.[1]

On the other hand Archmage logic is capable of verifying the correctness of xor-lists by virtue of the flexibility that Archmage provides in between logical and physical pointer representations. To illustrate this fact, in this paper we will focus on the correctness of a function that deletes an item from the head of an xor-list `delete_hd`, whose exact implementation is provided in Fig. 6.2. The xor-list node contains two integers, (*i*) `item`, which contains the value stored in the list, and (*ii*) `link`, which contains xor prv nxt as discussed in the xor-list traversal method. In `delete_hd`, the arguments hdH and tlH are pointers to memory locations that store pointers to the head and tail nodes of the xor-list, respectively. The full xor-list implementation and the proof of correctness for each of the functions may be found as part of our Coq development [21]; we

[1]The Quasi-Concrete model [4]. or PNVI-ae-udi [5], can support xor-lists, but do not come with a program logic.

XORLIST SPEC

$$
\begin{array}{l}
\texttt{def frag}_q\ \mathsf{m_{prv}}\ \mathsf{m_{nxt}}\ (\mathsf{prv\ hd\ tl\ nxt} : \mathsf{Val})\ (\mathsf{xs} : \texttt{list}\ \mathsf{Val}) := \\
\quad \texttt{match xs with} \\
\quad \mid [] \Rightarrow \mathsf{prv} \approx^{\mathsf{m_{prv}}} \mathsf{tl} * \mathsf{nxt} \approx^{\mathsf{m_{nxt}}} \mathsf{hd} \\
\quad \mid \mathsf{x} :: \mathsf{xs}' \Rightarrow \exists\, \mathsf{m_{hd}}\ i_{\mathsf{prv}}\ i_{\mathsf{mid}}, \\
\qquad\qquad \ulcorner \mathsf{m_{hd}}.sz = 2 \urcorner * \mathsf{prv} \approx^{\mathsf{m_{prv}}} i_{\mathsf{prv}} * \mathsf{live}_q^{\mathsf{m_{hd}}}(\mathsf{hd}) * \mathsf{hd} \mapsto_q^{\mathsf{m_{hd}}} [\mathsf{x}, \mathsf{xor}\ i_{\mathsf{prv}}\ i_{\mathsf{mid}}] * \\
\qquad\qquad \texttt{frag}_q\ \mathsf{m_{hd}}\ \mathsf{m_{nxt}}\ \mathsf{hd}\ i_{\mathsf{mid}}\ \mathsf{tl}\ \mathsf{nxt}\ \mathsf{xs}' \\
\quad \texttt{end} \\
\texttt{def xorlist}_q\ (\mathsf{hdH\ tlH} : \mathsf{Val})\ (\mathsf{xs} : \texttt{list}\ \mathsf{Val}) := \exists\, \mathsf{m_{hdH}}\ \mathsf{m_{tlH}}\ \mathsf{hd\ tl}, \\
\quad \mathsf{hdH} \mapsto_q^{\mathsf{m_{hdH}}} \mathsf{hd} * \mathsf{tlH} \mapsto_q^{\mathsf{m_{tlH}}} \mathsf{tl} * \texttt{frag}_q\ \mathsf{m_{NULL}}\ \mathsf{m_{NULL}}\ \texttt{NULL}\ \mathsf{hd\ tl}\ \texttt{NULL}\ \mathsf{xs}
\end{array}
$$

PRE & POST CONDITIONS

$$
\begin{array}{r}
\{\, \texttt{xorlist}_1\ \mathsf{hdH\ tlH\ xs} \,\}\ \texttt{add\_hd(hdH, tlH, x)}\ \{\, \texttt{xorlist}_1\ \mathsf{hdH\ tlH}\ (\mathsf{x} :: \mathsf{xs}) \,\} \\
\{\, \texttt{xorlist}_1\ \mathsf{hdH\ tlH\ xs} \,\}\ \texttt{add\_tl(hdH, tlH, x)}\ \{\, \texttt{xorlist}_1\ \mathsf{hdH\ tlH}\ (\mathsf{xs} +\!\!+ [\mathsf{x}]) \,\} \\
\{\, \texttt{xorlist}_1\ \mathsf{hdH\ tlH}\ (\mathsf{x} :: \mathsf{xs}) \,\}\ \texttt{delete\_hd(hdH, tlH)}\ \{\, r.\ \ulcorner r = \mathsf{x} \urcorner * \texttt{xorlist}_1\ \mathsf{hdH\ tlH\ xs} \,\} \\
\{\, \texttt{xorlist}_1\ \mathsf{hdH\ tlH}\ (\mathsf{xs} +\!\!+ [\mathsf{x}]) \,\}\ \texttt{delete\_tl(hdH, tlH)}\ \{\, r.\ \ulcorner r = \mathsf{x} \urcorner * \texttt{xorlist}_1\ \mathsf{hdH\ tlH\ xs} \,\}
\end{array}
$$

Figure 6.3: Correctness specifications for the functions `add_hd`, `add_tl`, `delete_hd`, and `delete_hd`.

focus on `delete_hd` to keep the presentation intuitive.

We start the verification of `delete_hd` by observing the correctness specification for `delete_hd`, as listed in the second box of Fig. 6.3. The specification is simple on a high level: given an xor-list of the form $\mathsf{x} :: \mathsf{xs}$ (encoded by the precondition $\texttt{xorlist}_1\ \mathsf{hdH\ tlH}\ (\mathsf{x} :: \mathsf{xs})$), `delete_hd` simply returns the removed value x and guarantees that x is removed from the xor-list.[2]

The definition of the predicate `xorlist` is given on the top of Fig. 6.3: intuitively, it checks that the locations pointed to by hdH and tlH store valid pointers to an xor-list containing the *values* xs in its `item` entries, with fractional permission q (the permission q is required when reading from the list, or in the case of `delete_hd`, we require $q = 1$ to make modifications to the list). Note that the correctness specification merely asks for the `item` entries, without exposing the address `link` entries: this corresponds with the user-level abstraction of a

[2]The implementation of `delete_hd` also allows the case in which an empty list is given, upon which `delete_hd` will return 0; here, we consider only non-empty lists for simplicity of presentation.

list, which is simply a traversable list of values.

More specifically, `xorlist` requires that (i) there exists some hd and tl that point to the head and tail of the xor-list, that hdH and tlH point to; and (ii) hd and tl can be traversed as an xor-list containing the values xs ($\texttt{frag}_q$ $\mathsf{m}_{\texttt{NULL}}$ $\mathsf{m}_{\texttt{NULL}}$ NULL hd tl NULL xs). The predicate `frag` is defined on the top of Fig. 6.3, and encodes the actual xor-list traversal methodology described at the start of this section—observe the case in which $\mathsf{xs} = \mathsf{x} :: \mathsf{xs}'$, in which `frag` requires that hd (*i.e.,* the current head) points to a valid node[3] with x as `item`, and xor i_{prv} i_{mid} as `link`, where prv must have a physical representation i_{prv}, and i_{mid} denotes the physical address of the next node to traverse (*i.e.,* $\texttt{frag}_q$ m_{hd} $\mathsf{m}_{\mathsf{nxt}}$ hd i_{mid} tl nxt xs′).

Having understood how the correctness specification is encoded, we proceed to show how one proves that `delete_hd` satisfies the correctness specification from Fig. 6.3. Fig. 6.4 lists the intermediate pre-postconditions that are generated at each step of the proof; we show how each statement in `delete_hd` can be shown to satisify these pre-postconditions using the rules of Archmage logic.

We focus on how going through each statement of `delete_hd` *changes* a given precondition into another postcondtion, and thus illustrate the proof by focusing on the transitions between predicates, which are numbered in Fig. 6.4 (❶ to ⓬). Changes in a postcondition compared to a precondition are highlighted in red.

We start with ❶, which is identical to the precondition of the correctness specification; ❶ to ❷ is a simple expansion of the definition of `xorlist`.

On the transition from ❷ to ❸, we have three loads, whose results are highlighted in red at the end of ❸, and are derivable by an application of the load rule. Note that the true-branching condition $\mathsf{hd\_old} \neq \texttt{NULL}$ follows from

[3]We denote $p \mapsto_q^{\mathsf{m}} [v_1, v_2]$ as shorthand for $p \mapsto_q^{\mathsf{m}} v_1 * p + \texttt{sizeof}(\mathsf{Val}) \mapsto_q^{\mathsf{m}} v_2$; *i.e., p* points to a pair of values v_1, v_2.

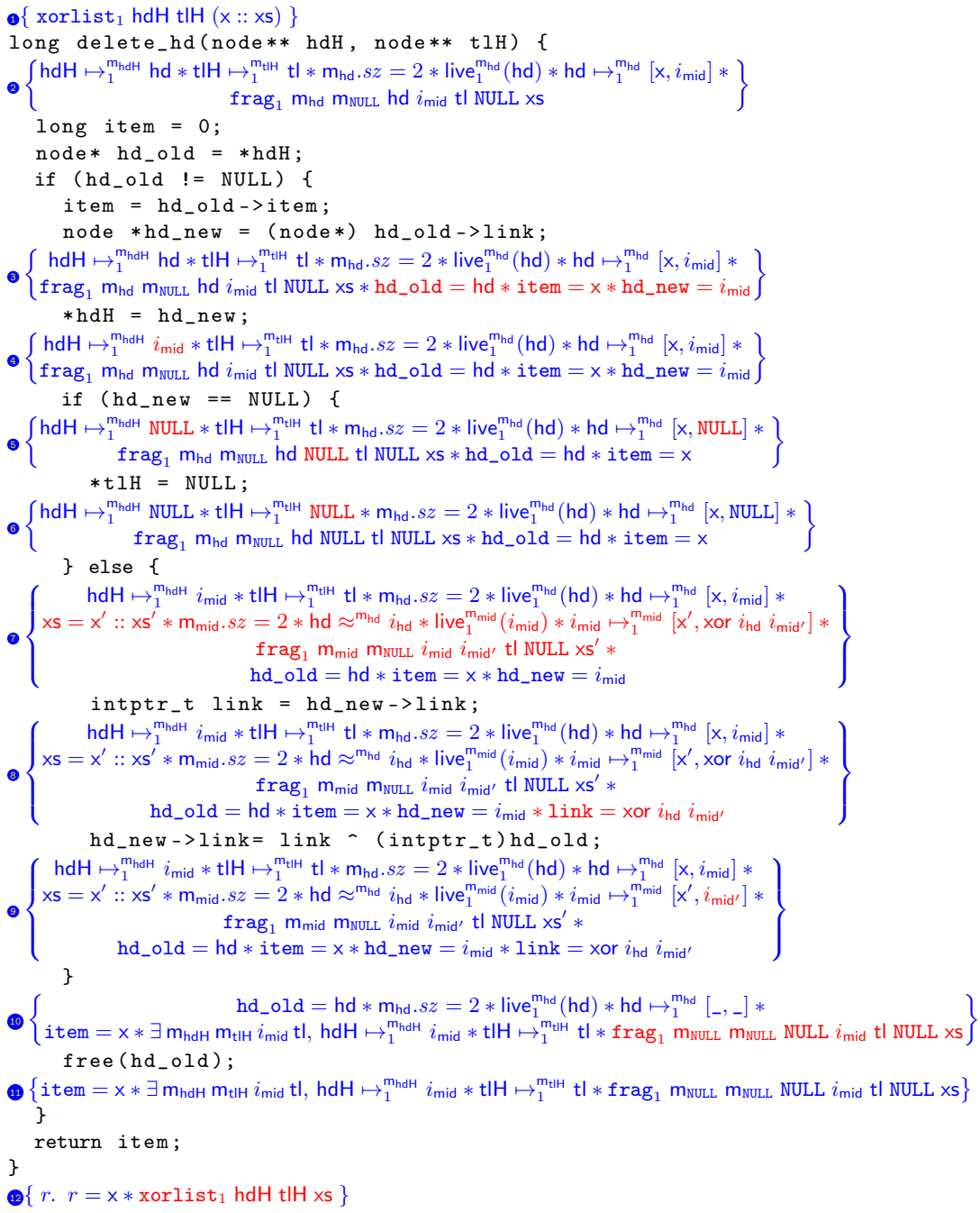

Figure 6.4: Pre- and postconditions generated at each step of the proof when verifying `delete_hd` in Archmage logic. The program code is black, the pre- and postconditions are blue, and changes introduced to the predicates at each step of the proof are highlighted in red.

$\mathsf{hd\_old} = \mathsf{hd}$ and $\mathsf{hd} \mapsto_1^{\mathsf{m_{hd}}} [\mathsf{x}, i_{\mathsf{mid}}]$.

The transition from ❸ to ❹ is simple: the statement is a store to hdH, and we simply use the store rule with the fact that $\mathsf{hd\_new} = i_{\mathsf{mid}}$ to obtain that $\mathsf{hdH} \mapsto_1^{\mathsf{m_{hdH}}} i_{\mathsf{mid}}$.

Moving forwards from ❹, we now encounter an if-statement. ❺ is the case for the *true* branch, which captures the case where x was the only item in the list and the list is now empty. Moving from ❹ to ❺, we simply update the predicate with the information that $\mathsf{hd\_new} = i_{\mathsf{mid}} = \texttt{NULL}$. Going from ❺ to ❻ is a write to tlH, which is simply updated via the store rule.

❼ is the case when the branch condition evaluates to *false* starting from ❹ (not ❻, as we are now analyzing the *false* branch). In ❼, the updated red predicate is simply an expansion of $\texttt{frag}_1\ \mathsf{m_{hd}}\ \mathsf{m}_{\texttt{NULL}}\ \mathsf{hd}\ i_{\mathsf{mid}}\ \mathsf{tl}\ \texttt{NULL}\ \mathsf{xs}$ on the second line of ❹, to the second case of frag as xs is non-empty (more precisely, one may drop the first case as now $i_{\mathsf{mid}} = \mathsf{hd\_new} \neq \texttt{NULL}$, thus the first case of frag is guaranteed to be *false*).

❼ to ❽ is simply a load statement whose result is stored in link; one may follow $\mathsf{hd\_new} = i_{\mathsf{mid}}$ and $i_{\mathsf{mid}} \mapsto_1^{\mathsf{m_{mid}}} [\mathsf{x}', \mathsf{xor}\ i_{\mathsf{hd}}\ i_{\mathsf{mid}'}]$ to obtain $\texttt{link} = \mathsf{xor}\ i_{\mathsf{hd}}\ i_{\mathsf{mid}'}$.

❽ to ❾ is a store operation that encodes the new link for the new head hd_new: because $\mathsf{hd\_old} = \mathsf{hd} \approx^{\mathsf{m_{hd}}} i_{\mathsf{hd}}$ and $\texttt{link} = \mathsf{xor}\ i_{\mathsf{hd}}\ i_{\mathsf{mid}'}$, the new value of hd_new->link $= i_{\mathsf{mid}'}$ as in ❾.

Having obtained the conclusion for both branches ❻ and ❾, we proceed to merge them as the condition ❿. The proof obligation is that ❻ $\implies$ ❿ and ❾ $\implies$ ❿; for ❻ $\implies$ ❿, $i_{\mathsf{mid}} = \mathsf{tl} = \texttt{NULL}$ in ❿ serves as witness for the implication to hold. In the case of ❾ $\implies$ ❿, the implication holds as the red $\texttt{frag}_1\ \mathsf{m}_{\texttt{NULL}}\ \mathsf{m}_{\texttt{NULL}}\ \texttt{NULL}\ i_{\mathsf{mid}}\ \mathsf{tl}\ \texttt{NULL}\ \mathsf{xs}$ in ❿ is the result of re-folding the predicates about i_{mid} on the second and third line of ❾, while $\mathsf{hd} \approx^{\mathsf{m_{hd}}} i_{\mathsf{hd}}$,

$\mathsf{hd\_new} = i_{\mathsf{mid}}$, $\mathtt{link} = \mathsf{xor}\ i_{\mathsf{hd}}\ i_{\mathsf{mid}'}$ and $\mathsf{xs} = \mathsf{x}' :: \mathsf{xs}'$ have been dropped.

Finally, the free statement consumes the resources for hd_old to yield ⓫; one may simply re-fold the definition of `xorlist` to obtain ⓬, the final desired postcondition.

In addition to the correctness of `delete_hd`, Archmage logic will also allow one to prove properties on `xorlist` as well, such as Theorem 2.

Theorem 2 (XorlistReverse)**.** *For any value* q, v_1, v_2, xs*:*

$$\mathit{xorlist}_q\ \mathsf{hdH}\ \mathsf{tlH}\ \mathsf{xs} \mathrel{*\!\!-\!\!*} \mathit{xorlist}_q\ \mathsf{tlH}\ \mathsf{hdH}\ \mathsf{reverse}(\mathsf{xs}).$$

Theorem 2 allows us to reuse the specifications listed in $\mathtt{frag}_q$ and $\mathtt{xorlist}_q$, from the top of Fig. 6.3, to prove the correctness of `delete_tl` as well. Without Theorem 2 the proof would be challenging, as the definition of, e.g., $\mathtt{frag}_q$ unfolds a list starting from the head, whereas `delete_tl` deletes an element from the tail.

Proof Size. To provide a measure of how effective Archmage logic is in proving programs with integer-pointer casts, we report the lines of code (LoC) required for fully mechanizing the correctness proof of the xor linked list discussed in this section.

```
#include "xorlist.h"
int main() {
  node *head = NULL, *tail = NULL;
  long item = 1;
  add_hd(&head, &tail, 3);
  add_tl(&head, &tail, 7);
  item *= delete_hd(&head, &tail);
  item *= delete_tl(&head, &tail);
  return item;
}
```

Figure 6.5: A code fragment illustrating a simple function `main` that is a client of the xor-list defined in `xorlist.h`.

Writing out the full specification of the xor linked list functions (`add_hd`, `add_tl`, `delete_hd`, `delete_tl`) took around 100 LoC, and defining additional lemmas required for the proof (such as reverse) took an additional 80 LoC. The proofs for each of these functions each took around 300 LoC for `add_hd` and `add_tl`, and 250 LoC for `delete_hd` and `delete_tl`, for a total of around 1300 LoC to specify and verify the xor linked list implementation.

```
// Encoding function
uintptr_t encode(long k, void *p) {
    uintptr_t encoded = (uintptr_t)p ^ k;
    return encoded;
}

// Decoding function
void *decode(long k, uintptr_t ep) {
    void *decoded = (void *)(ep ^ k);
    return decoded;
}
```

```
// Function that uses encoded pointer
long bar(long k, uintptr_t ep, long x) {
    long *q = decode(k, ep);
    *q = x;
    return *q;
}

// Function that creates encoded pointer
long foo(long *p, long k, long x) {
    uintptr_t ep = encode(k, p);  // pointer encoding
    bar(k, ep, x);     // pass encoded pointer
    return *p;  // *p = x
}
```

Figure 6.6: Snippets of Simplified Pointer Hardening Example.

In addition to verifying the correctness of the xor linked list itself, we are also interested in whether verifying a client that calls the xor-list as a library is also possible and efficient. Fig. 6.5 depicts a small function `main` that uses the xor-list; verifying `main` took around an additional 250 LoC.

Considering that the implementation of the xor linked list is lines about 70 lines in C, we conclude that Archmage logic provides a effective, scalable method of verifying source-level programs with integer-pointer casts.

6.3 Case Study 2: Proving Correctness of a Simple Pointer Hardening with Archmage Logic

In this section, we will show the correctness of simple pointer hardening example with Archmage Logic. Pointer hardening is a memory address protection technique used in low-level system programming to defend pointers from attackers. A notable implementation can be found in Linux's SLUB allocator [25], which applies pointer hardening to protect memory addresses of freed objects stored in its freelist. Fig. 6.6 shows a simple pointer hardening example where function `foo` passes three arguments to function `bar`: an encoded pointer, its encoding key k, and item `x`. Function `bar` then uses encoding key k to decode the pointer and stores item `x` at the decoded address. Although this example is

PRE & POST CONDITIONS

$$\{\, \mathsf{live}^{\mathsf{m}}_{q}(\mathsf{p} - \mathit{ofs}) \,\}\ \texttt{encode(k, p)}\ \{\, r.\ \ulcorner r = \mathsf{xor\ ip\ k} \urcorner * \mathsf{live}^{\mathsf{m}}_{q}(\mathsf{p} - \mathit{ofs}) * \mathsf{p} \approx^{\mathsf{m}} \mathsf{ip} \,\}$$

$$\{\,\}\ \texttt{decode(k, ep)}\ \{\, r.\ \ulcorner r = \mathsf{xor\ ep\ k} \urcorner \,\}$$

$$\{\, \mathsf{p} \approx^{\mathsf{m}} \mathsf{ip} * \mathsf{p} \mapsto^{\mathsf{m}}_{1} v * \ulcorner \mathsf{ep} = \mathsf{xor\ ip\ k} \urcorner \,\}\ \texttt{bar(k, ep, x)}\ \{\, r.\ \ulcorner r = \mathsf{x} \urcorner * \mathsf{p} \mapsto^{\mathsf{m}}_{1} \mathsf{x} \,\}$$

$$\{\, \mathsf{p} \mapsto^{\mathsf{m}}_{1} v * \mathsf{live}^{\mathsf{m}}_{1}(\mathsf{p} - 0) \,\}\ \texttt{foo(p, k, x)}\ \{\, r.\ \ulcorner r = \mathsf{x} \urcorner * \mathsf{p} \mapsto^{\mathsf{m}}_{1} \mathsf{x} * \mathsf{live}^{\mathsf{m}}_{1}(\mathsf{p} - 0) \,\}$$

Figure 6.7: Correctness specifications for the functions `encode`, `decode`, `bar`, and `foo`.

very simple, it employs similar subtle reasoning as used in proving the xor-list example, because (i) pointer `p` must be cast to an integer to perform bitwise-xor with encoding key k, and (ii) the encoded pointer must be reproduced as a pointer to write item x in the memory location pointed to by original pointer `p`.

We will show that both `foo` and `bar` satisfy the Hoare triple style specifications shown in Fig. 6.7. Fig. 6.8 lists the intermediate pre- and postconditions generated at each proof step. We will explain how each statement in `foo` and `bar` satisfies these conditions using the rules of Archmage logic (we will omit the proof of specifications for the decode and encode functions). As this example is simpler than the XOR-based linked list example, we will provide a more detailed explanation.

In fig. 6.8, we explain the proof of each function by focusing on the transitions between numbered predicates. Changes in the postcondition compared to the precondition are highlighted in red.

First, we focus on `bar`. Note that the original pointer `p` before encoding is not included in the arguments of `bar`. ❶ of `bar` is same as precondition of the Hoare triple of `bar`. The translation from ❶ to ❷ is simple. the statement is a call `decode` function, and we simply use hoare style specification of `decode`. This rule adds predicate about physical address of decoded pointer ($q = \mathsf{xor\ ep\ k}$) in postcondition.

❶{ $\mathsf{p} \approx^{\mathsf{m}} \mathsf{ip} * \mathsf{p} \mapsto_1^{\mathsf{m}} v * \mathsf{ep} = \mathsf{xor}\ \mathsf{ip}\ \mathsf{k}$ }
```
long bar(long k, uintptr_t ep, long x) {
  long *q = decode(k, ep);
```
❷{$\mathsf{p} \approx^{\mathsf{m}} \mathsf{ip} * \mathsf{p} \mapsto_1^{\mathsf{m}} v * \mathsf{ep} = \mathsf{xor}\ \mathsf{ip}\ \mathsf{k} * \mathsf{q} = \mathsf{xor}\ \mathsf{ep}\ \mathsf{k}$}
❸{$\mathsf{p} \approx^{\mathsf{m}} \mathsf{q} * \mathsf{p} \mapsto_1^{\mathsf{m}} v$}
❹{$\mathsf{p} \approx^{\mathsf{m}} \mathsf{q} * \mathsf{q} \mapsto_1^{\mathsf{m}} v$}
```
  *q = x;
```
❺{$\mathsf{p} \approx^{\mathsf{m}} \mathsf{q} * \mathsf{q} \mapsto_1^{\mathsf{m}} \mathsf{x}$}
```
  return *q;
}
```
❻{ $r.\ \ulcorner r = \mathsf{x} \urcorner * \mathsf{p} \mapsto_1^{\mathsf{m}} \mathsf{x}$ }

❶{ $\mathsf{p} \mapsto_1^{\mathsf{m}} v * \mathsf{live}_1^{\mathsf{m}}(\mathsf{p} - 0)$ }
```
long foo(long* p, long k, long x) {
  *p = 0;
```
❷{$\mathsf{p} \mapsto_1^{\mathsf{m}} 0 * \mathsf{live}_1^{\mathsf{m}}(\mathsf{p} - 0)$}
```
  uintptr_t ep = encode(k, p);
```
❸{$\mathsf{p} \mapsto_1^{\mathsf{m}} 0 * \mathsf{live}_1^{\mathsf{m}}(\mathsf{p} - 0) * \mathsf{p} \approx^{\mathsf{m}} \mathsf{ip} * \mathsf{ep} = \mathsf{xor}\ \mathsf{ip}\ \mathsf{k}$}
```
  bar(k, ep, x);
```
❹{$\mathsf{p} \mapsto_1^{\mathsf{m}} \mathsf{x} * \mathsf{live}_1^{\mathsf{m}}(\mathsf{p} - 0) * \mathsf{p} \approx^{\mathsf{m}} \mathsf{ip} * \mathsf{ep} = \mathsf{xor}\ \mathsf{ip}\ \mathsf{k}$}
```
  return *p;
}
```
❺{ $r.\ \ulcorner r = \mathsf{x} \urcorner * \mathsf{p} \mapsto_1^{\mathsf{m}} \mathsf{x} * \mathsf{live}_1^{\mathsf{m}}(\mathsf{p} - 0)$ }

Figure 6.8: Pre- and postconditions generated at each step of the proof when verifying `foo` and `bar` in Archmage logic. The program code is black, the pre- and postconditions are blue, and changes introduced to the predicates at each step of the proof are highlighted in red.

The transition from ❷ to ❸ checks that q is the physical address of p ($\mathsf{p} \approx^{\mathsf{m}} \mathsf{q}$). To prove this, we follow several steps. First, we substitute $\mathsf{ep} = \mathsf{xor}\ \mathsf{ip}\ \mathsf{k}$ into $\mathsf{q} = \mathsf{xor}\ \mathsf{ep}\ \mathsf{k}$, yielding $\mathsf{q} = \mathsf{xor}\ (\mathsf{xor}\ \mathsf{ip}\ \mathsf{k})\ \mathsf{k}$. Next, by applying the properties of XOR operations, we can deduce that $\mathsf{q} = \mathsf{ip}$. Finally, substituting $\mathsf{q} = \mathsf{ip}$ into $\mathsf{p} \approx^{\mathsf{m}} \mathsf{ip}$ (which appears in the precondition) establishes $\mathsf{p} \approx^{\mathsf{m}} \mathsf{q}$ in the postcondition.

Moving from ❸ to ❹, we apply rule (8) from fig. 6.1 to the precondition $\mathsf{p} \approx^{\mathsf{m}} \mathsf{q} * \mathsf{p} \mapsto_1^{\mathsf{m}} v$. This rule establishes that once we have proven q is the physical address of p, we can conclude that q has access permission to the same memory location that p points to.

The transition from ❹ to ❺ represents a simple store operation that writes x to the location pointed to by q.

The final transition from ❺ to ❻ involves a load operation that reads the value at q and returns it. Since x was previously stored at q, the return value is x. We then apply rule (8) from fig. 6.1 in the reverse direction compared to the ❸-to-❹ transition, which restores the points-to predicate back to pointer p.

Significantly, although function `bar` does not receive pointer p as an argument, we can establish that q is the physical address of p ($\mathsf{p} \approx^{\mathsf{m}} \mathsf{q}$), enabling access to the same memory location through the decoded pointer q.

We now examine the specification for function `foo`, which calls `bar`. ❶ represents `foo`'s precondition, which requires two conditions: the accessibility of the original pointer p ($\mathsf{p} \mapsto_1^{\mathsf{m}} v$) and its liveness ($\mathsf{live}_1^{\mathsf{m}}(\mathsf{p} - 0)$). The transition from ❶ to ❷ represents a simple store operation that writes zero to the memory location pointed to by p.

Moving from ❷ to ❸, the code executes an encoding function that encodes pointer p using key k. This encoding function accepts a pointer as its argument and performs two operations: pointer-to-integer casting followed by a bitwise-

XOR operation. Its precondition requires the liveness of the original pointer, while its postcondition establishes two new predicates: the relation between the original pointer and its corresponding physical address ($\mathsf{p} \approx^{\mathsf{m}} \mathsf{ip}$), and the value of the encoded pointer ($\mathsf{ep} = \mathsf{xor}\ \mathsf{ip}\ \mathsf{k}$).

❸ advances to ❹ through a function call to `bar`. The `bar` function performs two operations: it decodes the encoded pointer and stores x at the resulting location. Since the decoded pointer represents the physical address of the original pointer, storing a value through the decoded pointer has the same effect as storing through the original pointer. `bar`'s precondition requires two elements: the information necessary for decoding the encoded pointer ep ($\mathsf{p} \approx^{\mathsf{m}} \mathsf{ip} * \mathsf{ep} = \mathsf{xor}\ \mathsf{ip}\ \mathsf{k}$) and permission to store a new value at the decoded pointer location ($\mathsf{p} \mapsto^{\mathsf{m}}_{1} 0$). Its postcondition captures the updated state, showing that x is now stored at the location that pointed to by original pointer p ($\mathsf{p} \mapsto^{\mathsf{m}}_{1} \mathsf{x}$).

Finally, ❹ and ❺, a load statement reads the value from the memory location pointed to by p and returns that value. Since the `bar` function previously stored x at this memory location, the return value is x.

Proof Size. Similar to the xor-list example, we quantify the implementation effort required to mechanize the correctness proof of the pointer hardening example discussed in this section.

The complete implementation required approximately 400 lines of code (LoC). The specification of pointer hardening functions took around 80 LoC, while the additional lemmas needed for the proof required approximately 60 LoC. The verification proofs varied in size across functions: approximately 50 LoC for `encode`, 40 LoC for `decode`, 70 LoC for `bar`, and 80 LoC for `foo`.

Chapter 7

Discussion and Related Works

In this section, we provide a detailed discussion of the capabilities of Archmage and CompCertCast with previous work on formalizing and verifying various aspects of C and its compilation, focusing on those that concern integer-pointer casts.

CompCertCast and the original CompCert backend One natural question that readers may have about CompCertCast is: how much does the restriction that physical memory consumption cannot be reduced affect the backend optimizations in CompCertCast? Here, we clarify that *all* existing optimiziations that are performed by original CompCert during compilation from the source (Clight) to the target (Asm) language, may be performed in CompCertCast as well. This is because CompCertCast preserves the original correctness principle of CompCert when performing optimizations: the compiler (both original CompCert and CompCertCast) will not reduce the consumption of "public" memory (memory blocks whose addresses may have been leaked to an external

function), and instead only perform consumption-reducing optimizations on "private" memory. This is because even in original CompCert, reducing public memory consumption may trigger undefined behavior in the target if the target attempts to access a memory region that has been optimized away.

We will illustrate this fact through SimplLocal, an example optimization pass in original CompCert and CompCertCast. SimplLocal is an optimization that moves scalar variables whose addresses have not been taken to local temporary storage. At first sight, this may seem to reduce memory consumption, because variables are moved from memory to temporary storage. However, SimplLocal does *not* move variables whose addresses *have* been taken, because these variables are considered to be in public memory in original CompCert. Variables which have been allocated a physical memory address in Archmage and CompCertCast are guaranteed to have their address taken—and thus in CompCertCast, SimplLocal is an optimization that only reduces consumption of logical memory, which can be justified by Archmage.

That said, while CompCertCast does allow one to perform all existing CompCert optimizations, there are certain cases in which optimizations are less effective when compared to original CompCert. One case is dead code elimination: CompCertCast is unable to remove dead *casts*, because removing a cast would reduce the usage of physical memory in the target. Another case is optimizations that rely on value analysis in CompCert: for example, common subexpression elimination (as discussed in §5.2); constant propagation and dead code elimination can also be less effective for similar reasons. Common subexpression elimination also does not work well for psub, if there exists, e.g., an unknown builtin function call between the to-be eliminated psubs.

Architecture-wise, CompCertCast is implemented only for x86-64 architectures, and not for other target architectures such as ARM or PowerPC.

Extending CompCertCast towards other architectures should not pose a theoretical challenge, but is mostly an implementation challenge. For example, ARM contains many variants of comparison, which would necessitate us to implement different versions of comparisons for each of these comparison operators.

CompCertS CompCertS [26, 12, 27] represents a different approach to supporting integer-pointer casts in CompCert, by extending the CompCert memory model with symbolic expression pointers to support integer-pointer casts. The main idea in CompCertS is to construct symbolic expressions whenever an operation takes place, instead of computing a concrete value. These symbolic expressions are only concretized on a by-need basis through a normalization function that relies on an SMT solver.

Because the underlying approaches are so different, CompCertS and Archmage-CompCertCast display a variety of differences: One notable difference is that CompCertCast supports more source-level integer-pointer casting patterns compared to CompCertS. In general, any program that displays nondeterminism when concretizing pointers according to the semantics of Archmage will be undefined in CompCertS [28]. One example of such a pattern would be a hash table that takes a pointer as a key. We do observe that CompCertS would be capable of verifying the xor-list example in §6, because upon access, when concretization occurs in the symbolic-value model, all xor-operations on pointers become canceled out.

Another major difference is in policies regarding memory usage: as previously discussed, CompCertCast follows the same policy as original CompCert and thus can preserve all backend optimizations. On the other hand, CompCertS uses a separate policy: memory usage must be not be *increased*, which is not simply a design choice but a necessity in the symbolic model of CompCertS [27].

In theory, this would prevent CompCertS from performing optimizations from original CompCert that increase memory usage. CompCertS circumvents this issue by computing and allocating the total additional memory usage it will require during the optimization phase prior to starting the compilation chain (so called memory provisioning). However, there are still some optimizations that do not fit both the memory preservation policy and memory provisioning, such as tail call optimizations and inlining, that would require future work on the symbolic model to support in CompCertS.

Other work on formalizing integer-pointer casts Stackaware CompCert [29] is an extension of CompCert that, while not directly related to integer-pointer casting, provides an interesting point of comparison to our lower bound improvement. The assembly generated by stackaware CompCert also closely that of machine code, in that the stacks of functions are coalesced into a single big stack, similar to how actual machine code will allocate function stacks by moving a stack pointer in memory. However, assembly generated by stackaware CompCert still contains logical pointers; the lower bound improvement presented in §5.3 thus represents an orthogonal improvement.

VST [2, 8] is a separation logic that targets a source language called verifiable C, which is a subset of C. VST provides an end-to-end verification scheme for verifiable C programs, packaged within a well-developed user interface; however, verifiable C does not support integer-pointer casts [30], and thus VST is incapable of supporting integer-pointer casts as well.

The Quasi-Concrete model [4] extends the CompCert memory model to support integer-pointer casts. While the Quasi-Concrete model is capable of supporting much of the coding patterns discussed in this paper, it does not support one key pattern: casting integers into one-past-the-end pointers. In

particular, being unable to cast integers into one-past-the-end pointers *prevents integers from refining pointers* in the Quasi-Concrete model. As discussed in §5, this is a very desirable property in the context of optimizations; indeed, the Quasi-Concrete model cannot justify optimizations such as cast propagation.

While not a memory model itself, the idea of *angelic nondeterminism* utilized in DimSum [31] offers an alternative way of creating a memory model that supports the casting of integers as one-past-the-end pointers. However, using angelic nondeterminism for integer-to-pointer casts prevents *reordering optimizations* between casts and other sources of nondeterminism (e.g., external calls), creating another missed opportunity for applying optimizations.

PNVI-ae-udi [5] is a formalization of C semantics that also formalizes integer-pointer casts through a memory model. Being a successful formalization, PNVI-ae-udi is indeed capable of supporting the example coding patterns and optimizations in this paper; however, PNVI-ae-udi is also highly complex, making it an undesirable tool for source-level verification.

VIP [10] is a "simplification" of the memory model of PNVI-ae-udi, in that VIP defines a simpler source-level semantics which PNVI-ae-udi refines. VIP especially enjoys automation through integration with RefinedC [32], enabling the automatic verification of some programs with integer-pointer casts. However, despite being a memory model targeted at source-level verification, VIP cannot support some coding patterns such as the xor-list. VIP also requires the source program to be annotated with a special instruction.

The Twin-Allocation model [11] is a memory model that formalizes integer-pointer casts in LLVM IR, and is capable of supporting many coding patterns and optimizations. However, the Twin-Allocation model (and VIP as well) do not formalize out-of-memory and simply assume that it does not happen, which is unsound in a formal end-to-end verification setting as in §4.

seL4 [33] is an approach that performs translation validation to formally verify an OS kernel, which, being low-level systems code, relies heavily on integer-pointer casts. The approach taken in seL4 represents another way one can establish end-to-end verification for code containing integer-pointer casts, through translation validation. However, while translation validation allows us to *check* that some emitted assembly is correct, it cannot provably *generate* correct assembly code given some arbitrary source program like CompCertCast can. In this paper, we mean end-to-end verification in the sense of a verified compiler.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This paper introduces a framework for end-to-end verification of C programs, specifically (*i*) Archmage, a memory model for integer-pointer casts, (*ii*) CompCertCast, an extension of CopmCert that supports integer-pointer casting and optimizations through Archmage, and (*iii*) Archmage logic, a source-level separation logic built on top of Archmage that supports reasoning about complex integer-pointer casting patterns. In particular, CompCertCast preserves the optimizations of original CompCert while introducing new optimizations to mitigate the overhead of formally considering integer-pointer casts, and Archmage logic represents a scalable source-level logic capable of verifying complex programs such as the xor-list in §6.2.

8.2 Future Work

Our work presents several opportunities for future research and development. This section outlines three key directions that could extend and improve our current findings.

First, there is future work needed to automate Archmage logic. We hope to extend the work presented in this thesis by automating Archmage logic to obtain a full verification chain capable of automatically verifying and compiling integer-pointer casting programs. As discussed in §7, related works such as VIP and RefinedC have successfully automated separation logic reasoning. Since CCR, which is the foundation of our program verification approach, can support most of the features of Iris [24] that are used in VIP and RefinedC, we believe similar automation is possible for Archmage logic.

Second, there are limitations in our approach that need to be addressed in future work. As discussed in §7, CompCertCast currently only supports x86-64 as a backend architecture, while the original CompCert supports multiple architectures including x86-32, x86-64, ARM, Power-PC, and RISC-V. Extending CompCertCast to support widely-used architectures like ARM would be one of the most efficient ways to expand Archmage's verification coverage. Additionally, as covered in §5, CompCertCast's RTL treats a function's stack as a single memory object. This stack structure makes it difficult to apply static analysis to stack-stored values in situations where the stack can be accessed from other functions (e.g., when a stack pointer is escaped to other function). While this is also an issue in CompCert, CompCertCast faces an additional problem: blocks with physical addresses can be accessed by other functions even if stack pointers haven't leaked externally. Therefore, we believe addressing this stack-related issue would provide greater benefits for CompCertCast compared to the original

CompCert.

Finally, there is a need to verify more complex examples involving integer-pointer casting. While Archmage currently handles examples like xor-list and simple pointer hardening, it was designed to verify more complex system programs. The SLUB allocator or KASAN [34] could be good targets for such verification. Since verification targets may use features that Archmage does not currently support, additional extensions to Archmage may be necessary.

Bibliography

[1] X. Leroy, "Formal certification of a compiler back-end or: Programming a compiler with a proof assistant," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2006, 2006.

[2] A. W. Appel, *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[3] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74, IEEE, 2002.

[4] J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis, "A formal c memory model supporting integer-pointer casts," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, 2015.

[5] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell, "Exploring c semantics and pointer provenance," *Proc. ACM Program. Lang.*, vol. 3, jan 2019.

[6] Y. Kim, M. Cho, J. Lee, J. Kim, T. Yoon, Y. Song, and C.-K. Hur, "Archmage and compcertcast: End-to-end verification supporting integer-pointer casting," *Proc. ACM Program. Lang.*, vol. 9, jan 2025.

[7] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "Certikos: An extensible architecture for building certified concurrent os kernels," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 2016, 2016.

[8] W. Mansky and K. Du, "An iris instance for verifying compcert c programs," *Proc. ACM Program. Lang.*, vol. 8, jan 2024.

[9] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, "Compcerttso: A verified compiler for relaxed-memory concurrency," *Journal of the ACM (JACM)*, vol. 60, no. 3, pp. 1–50, 2013.

[10] R. Lepigre, M. Sammler, K. Memarian, R. Krebbers, D. Dreyer, and P. Sewell, "Vip: Verifying real-world c idioms with integer-pointer casts," *Proc. ACM Program. Lang.*, vol. 6, jan 2022.

[11] J. Lee, C.-K. Hur, R. Jung, Z. Liu, J. Regehr, and N. P. Lopes, "Reconciling high-level optimizations and low-level code in llvm," *Proc. ACM Program. Lang.*, vol. 2, oct 2018.

[12] F. Besson, S. Blazy, and P. Wilke, "A Concrete Memory Model for CompCert," in *6th International Conference on Interactive Theorem Proving*, ITP 2015, 2015.

[13] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and*

Runtime Optimization, CGO '04, (USA), p. 75, IEEE Computer Society, 2004.

[14] The Coq Development Team, "The Coq proof assistant 8.13.2 reference manual," 2021. `https://coq.github.io/doc/V8.13.2/refman/`.

[15] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, (New York, NY, USA), p. 259–270, Association for Computing Machinery, 2005.

[16] J. Boyland, "Checking interference with fractional permissions," in *Static Analysis* (R. Cousot, ed.), (Berlin, Heidelberg), pp. 55–72, Springer Berlin Heidelberg, 2003.

[17] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, (USA), p. 55–74, IEEE Computer Society, 2002.

[18] Y. Song, M. Cho, D. Kim, Y. Kim, J. Kang, and C.-K. Hur, "Compcertm: Compcert with c-assembly linking and lightweight modular verification," *Proc. ACM Program. Lang.*, vol. 4, Dec. 2019.

[19] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, p. 451490, oct 1991.

[20] G. Barthe, D. Demange, and D. Pichardie, "Formal verification of an ssa-based middle-end for compcert," *ACM Trans. Program. Lang. Syst.*, vol. 36, mar 2014.

[21] Kim, Yonghyun and Cho, Minki and Lee, Jaehyung and Kim, Jinwoo and Yoon, Taeyoung and Song, Youngju and Hur, Chung-Kil, "Archmage artifact," 2024. `https://github.com/snu-sf/Archmage/`.

[22] L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic, "Interaction trees: Representing recursive and impure programs in coq," *Proc. ACM Program. Lang.*, vol. 4, Dec. 2019.

[23] Y. Song, M. Cho, D. Lee, C.-K. Hur, M. Sammler, and D. Dreyer, "Conditional contextual refinement," *Proc. ACM Program. Lang.*, vol. 7, jan 2023.

[24] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of Functional Programming*, vol. 28, 2018.

[25] L. Torvalds *et al.*, "Linux kernel," 2024.

[26] F. Besson, S. Blazy, and P. Wilke, "A precise and abstract memory model for c using symbolic values," in *Asian Symposium on Programming Languages and Systems*, 2014.

[27] F. Besson, S. Blazy, and P. Wilke, "CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics," in *Journal of Automated Reasoning*, 2019.

[28] J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis, "A formal c memory model supporting integer-pointer casts," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 326–335, 2015.

[29] Y. Wang, P. Wilke, and Z. Shao, "An abstract stack based approach to verified compositional compilation to machine code," *Proc. ACM Program. Lang.*, vol. 3, jan 2019.

[30] A. W. Appel, L. Beringer, Q. Cao, and J. Dodds, *Verifiable C*, vol. 1. 2023.

[31] M. Sammler, S. Spies, Y. Song, E. D'Osualdo, R. Krebbers, D. Garg, and D. Dreyer, "Dimsum: A decentralized approach to multi-language semantics and verification," *Proc. ACM Program. Lang.*, vol. 7, jan 2023.

[32] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. 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, (New York, NY, USA), p. 158–174, Association for Computing Machinery, 2021.

[33] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *SOSP*, pp. 207–220, ACM, 2009.

[34] D. Vyukov *et al.*, "KernelAddressSANitizer (KASAN)." `https://www.kernel.org/doc/html/latest/dev-tools/kasan.html`, 2014. Linux kernel memory error detector.

Acknowledgements

가장 먼저 저의 지도교수님이신 허충길 교수님께 감사드립니다. 집요하게 문제의 본질을 파악하는 자세와 동료들과 의견을 나누어야 한다는 가르침은 앞으로의 제 연구에서 가장 염두에 두어야 할 배움입니다. 교수님의 연구에 대한 열정적인 태도는 이따금씩 연구에 지쳐 있는 저에게 큰 격려가 되었습니다.

이광근 교수님께 감사드립니다. 교수님께서는 저에게 필요한 조언과 가르침을 아낌없이 베풀어 주셨습니다. 특히 발표에 있어서 교수님께서 주셨던 많은 조언들 덕분에 박사과정 중 중요한 고비에 있었던 발표들을 잘 헤쳐 나갈 수 있었습니다.

강지훈 교수님께 감사드립니다. 처음에는 연구실 선배로, 졸업한 이후에는 같은 분야를 연구하는 교수님으로 연구와 인생을 이끌어가는데 피와 살이 되는 가르침을 주셨습니다. 제가 박사 과정 동안 진행했던 대부분의 연구는 강지훈 교수님의 연구에 뿌리를 두고 있습니다. 교수님의 연구가 단단한 뿌리가 되어 주었기 때문에 저의 연구가 앞으로 나아갈 수 있었습니다.

김지응 교수님께 감사드립니다. 교수님께서는 프로그래밍 언어 분야의 선배로서도, 같은 주제를 연구하는 동료로서도 가장 좋은 분입니다. 저의 장래를 본인의 것처럼 같이 고민해 주시던 일을 절대 잊을 수 없을 것입니다.

이재욱 교수님께 감사드립니다. 교수님께서 주신 조언들 덕분에 생각하지도 못했던 새로운 예제들을 찾아보고 시야를 넓힐 수 있었습니다.

조민기에게 감사합니다. 민기가 놀라운 이해력과 분석력으로 제 연구에 기여한

바는 이루 다 말할 수 없고 그의 명랑한 성격이 아니었다면 저의 연구가 어려움에 빠졌을 때 다시 일어날 수 없었을 것입니다.

소프트웨어 원리 연구실과 프로그래밍 연구실의 동료분들께 감사드립니다. 이렇게 훌륭한 동료들과 함께 연구할 수 있는 환경에 있었다는 것이 제 인생에 있어서 큰 행운이었습니다. 특히 강지훈 교수님, 김윤승 형, 이준영, 송용주, 조민기, 이성환, 김동주, 김진우, 이재형에게 감사합니다. 함께 연구할 때 큰 힘이 되어주었던 김윤승 형, 조민기, 이재형, 김진우, 윤태영, 송용주, 김동주, 이동재에게 감사합니다. 연구실 생활을 함께 하며 대학원 과정 중 있었던 즐거움과 어려움을 함께 나눴던 김윤승 형, 이준영, 송용주, 신동연 형, 이동권, 김세훈, 이성환, 조민기, 김동주, 주호영, 이동재에게 감사합니다.

대학원 과정을 잘 마무리할 수 있게 도와주신 부모님께 감사드립니다. 두 분의 지원 덕분에 다른 걱정을 하지 않고 박사 과정을 마칠 수 있었습니다. 이것이 얼마나 큰 사랑인지 알고 있습니다.

대학원 밖에서 만났던 친구들에게 감사합니다. 대학에서 만난 뒤로 저에게 많은 위로와 격려를 주었던 권순형, 박지원, 김지원, 김도현에게 감사합니다. 저에게 프로그래밍을 본격적으로 소개하고 가르쳐주어 전산학에 발을 들이는데 도움을 준 차동훈에게 감사합니다.

마지막으로, 아내에게 이 논문을 바칩니다. 아내는 저의 대학원 생활 동안 등불과도 같은 존재였습니다. 아내의 도움이 없었다면 저는 박사 과정을 무사히 마칠 수 없었을 것입니다. 앞으로 아내가 저에게 주었던 큰 사랑을 갚아나가며 살겠습니다.

요약

본 논문은 정수-포인터 변환한 C 프로그램을 처음부터 끝까지 검증하는 방법을 제시한다. 기존의 정수-포인터 변환을 다루는 엄밀한 메모리 모델과 관련된 접근법은 처음부터 끝까지 검증하는 것을 고려하지 않고 디자인 되었기 때문에 다음과 같은 결점을 가질 수 있다. 첫째, 중요한 프로그래밍 패턴을 지원하지 않거나 둘째, 컴파일 과정의 일부를 설명하지 못하거나 셋째, 프로그램 검증을 위한 프로그램 논리를 지원하지 않는다.

본 논문에서는 정수-포인터 변환을 포함한 프로그램을 처음부터 끝까지 검증하기 위해 디자인된 프레임워크 Archmage을 소개한다. Archmage은 넓은 범위의 프로그래밍 패턴을 지원하며, 컴파일러의 많은 최적화를 설명할 수 있고, 프로그램 검증을 위한 프로그램 논리를 제공한다. Archmage은 두 개의 시스템으로 구성되어있다. 첫번째로 CompCertCast은 정수-포인터 변환을 지원하도록 확장된 CompCert로 정수-포인터 변환을 포함한 프로그램이 올바르게 컴파일된다는 것을 보장한다. 두번째로 Archmage logic은 정수-포인터 변환과 관련된 증명을 다룰 수 있는 프로그램 논리로, 정수-포인터 변환을 포함한 프로그램이 프로그래머가 정의한 명세를 따른다는 것을 증명하는 도구를 제공한다. 우리는 CompCertCast를 구현하면서 정수-포인터 변환을 지원하게 되며 생기는 성능 저하를 최소화하고 xor-list를 포함한 예제들을 Archmage logic으로 검증하여 이 프레임워크의 유용성을 보였다.

주요어: 정수-포인터 변환, 컴파일러, CompCert, 분리 논리, Coq, 정형 검증
학번: 2017-22945