Archmage and CompCertCast: End-to-End Verification Supporting Integer-Pointer Casting

YONGHYUN KIM, Seoul National University, South Korea
MINKI CHO*, Seoul National University, South Korea
JAEHYUNG LEE, Seoul National University, South Korea
JINWOO KIM, Seoul National University, South Korea
TAEYOUNG YOON, Seoul National University, South Korea
YOUNGJU SONG†, MPI-SWS, Germany
CHUNG-KIL HUR‡, Seoul National University, South Korea

Although there have been many approaches for developing formal memory models that support integer-pointer casts, previous approaches share the drawback that they are not designed for *end-to-end verification*, failing to support some important source-level coding patterns, justify some backend optimizations, or lack a source-level logic for program verification.

This paper presents Archmage, a framework for integer-pointer casting designed for end-to-end verification, supporting a wide range of source-level coding patterns, backend optimizations, and a formal notion of out-of-memory. To facilitate end-to-end verification via Archmage, we also present two systems based on Archmage: CompCertCast, an extension of CompCert with Archmage to bring a full verified compilation chain to integer-pointer casting programs, and Archmage logic, a source-level logic for reasoning about integer-pointer casts. We design CompCertCast such that the overhead from formally supporting integer-pointer casts is mitigated, and illustrate the effectiveness of Archmage logic by verifying an xor-based linked-list implementation, Together, our paper presents the first practical end-to-end verification chain for programs containing integer-pointer casts.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Formal software verification**; • **Theory of computation** → **Logic and verification**; **Separation logic**.

Additional Key Words and Phrases: Integer-Pointer Casting, Compiler, CompCert, Separation Logic, Coq, Verification

ACM Reference Format:
Yonghyun Kim, Minki Cho, Jaehyung Lee, Jinwoo Kim, Taeyoung Yoon, Youngju Song, and Chung-Kil Hur. 2025. Archmage and CompCertCast: End-to-End Verification Supporting Integer-Pointer Casting. *Proc. ACM Program. Lang.* 9, POPL, Article 45 (January 2025), 29 pages. https://doi.org/10.1145/3704881

*He contributed to this work while at Seoul National University. He is currently at FuriosaAI.
†He contributed to this work while at MPI-SWS. He is currently at Amazon.
‡Corresponding author.

Authors' Contact Information: Yonghyun Kim, Seoul National University, Seoul, South Korea, yonghyun.kim@sf.snu.ac.kr; Minki Cho, Seoul National University, Seoul, South Korea, minki.cho@sf.snu.ac.kr; Jaehyung Lee, Seoul National University, Seoul, South Korea, jaehyung.lee@sf.snu.ac.kr; Jinwoo Kim, Seoul National University, Seoul, South Korea, jinwoo.kim@sf.snu.ac.kr; Taeyoung Yoon, Seoul National University, Seoul, South Korea, taeyoung.yoon@sf.snu.ac.kr; Youngju Song, MPI-SWS, Saarbrucken, Germany, youngju@mpi-sws.org; Chung-Kil Hur, Seoul National University, Seoul, South Korea, gil.hur@sf.snu.ac.kr.

ACM 2475-1421/2025/1-ART45
https://doi.org/10.1145/3704881

1 Introduction

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 [Leroy 2006]—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 [Besson et al. 2015; Kang et al. 2015; Lee et al. 2018; Lepigre et al. 2022; Memarian et al. 2019] to formalize and verify integer-pointer casting outside of CompCert as well. For example, PNVI-ae-udi [Memarian et al. 2019] 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 [Kang et al. 2015] 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 [Lee et al. 2018] was developed in order to semi-formally justify pointer-related optimizations that are performed by LLVM [Lattner and Adve 2004], 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 [Lepigre et al. 2022] 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 (§4): 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 (§5). 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 §5, 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 (§3).
- 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 (§4).
- 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 (§5).

§2 provides a high-level overview of the entire Archmage framework, with a focus on new contributions. §6 discusses related work; and §7 concludes. Archmage, Archmage logic, and CompCertCast are all implemented using the Coq proof assistant [The Coq Development Team 2021]. Together, they provide the first full realization of a end-to-end verification pipeline with support for integer-pointer casts.

2 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.

2.1 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 [Besson et al. 2015; Kang et al. 2015; Lee et al. 2018; Lepigre et al. 2022; Memarian et al. 2019], and in particular has many similarities to the Quasi-Concrete model [Kang et al. 2015]. 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. 1 illustrates one of the reasons one-past-the-end pointers are challenging to support: their *ambiguity*. In Fig. 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.

```
1 char a[4], b[4];
2 i = (intptr_t) a;
3 j = (intptr_t) b;
4 if (i + 4 == j) {
5   p = (char *) (i + 4);
6   *(p - 1) = 42;
7   q = (char *) j;
8   *q = 37;
9 }
```

Fig. 1. A code fragment illustrating integer-pointer casting with one-past-the-end pointers.

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 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. 1), the behavior of an operation, such as memory accesses

[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.

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 3.1). 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 (§3).

2.2 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 §4. 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 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. 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. 2). Then, because the lifetime of p is up to the right-hand side of line 3,

[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).

```
1 foo () {
2   p = malloc(sizeof(int));
3   i = (int) p;
4   bar(p);
5   tar(i);
6   bar(p);
7 }
```

```
1 foo () {
2   p = malloc(sizeof(int));
3   i = (int) p;
4   bar(i);
5   tar(i);
6   bar(i);
7 }
```

```
1 foo () {
2   EBX = malloc(sizeof(int));
3   EBX = (int) EBX;
4   bar(EBX);
5   tar(EBX);
6   bar(EBX);
7 }
```

Fig. 2. An example of decreasing register pressure via applying cast propagation.

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. 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 §4.

2.3 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,
- $p \mapsto^{\mathsf{m}}_{q} v$, indicating that a pointer p points to a location containing the value v,
- $\mathsf{live}^{\mathsf{m}}_{q}(p)$, 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 $p \mapsto^{\mathsf{m}}_{1} v$ (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 [Bornat et al. 2005; Boyland 2003] for more 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. 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 $a + 1 \mapsto^{\mathsf{a}}_{1} \mathit{undef}$ (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 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 $i + 1 = p$),
- $p \mapsto^{\mathsf{a}}_1 \mathit{undef}$ (replacing $a + 1$ with p in $a + 1 \mapsto^{\mathsf{a}}_1 \mathit{undef}$).[3]

As line 5 can be proven to have write permissions to p, it can generate the postcondition $p \mapsto^{\mathsf{a}}_1 42$.

Lines 6-8 of Fig. 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 $p \mapsto^{\mathsf{a}}_1 42$ are defined over different block data (*i.e.,* a and b) and thus the substitution principle does not apply.

```
1 foo () {
2   char a[4], b[4];
3   i = (intptr_t) a;
4   p = (char *) (i + 1);
5   *p = 42;
6   j = (intptr_t) b;
7   if (i + 4 == j)
8     *(b - 3) = 37;
9   return (p == a + 1)
10 }
```

Fig. 3. A code fragment to illustrate the use of Archmage logic.

The third predicate $\mathsf{live}^{\mathsf{m}}_q(p)$ 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, one can prove that p == a + 1 at line 9 evaluates 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.,* $0 \le 1 \le \mathsf{a}.sz$), 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 §5.2. 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 by [Reynolds 2002] operates on a flat memory model, making it less suitable for compiler optimizations).

[3]Such replacements are only possible if the block in question a is identical; such details are formalized in §5.

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

Fig. 4. Type definitions for Archmage.

3 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).

Fig. 4 contains the definitions for the various constructs required for Archmage, and Fig. 5 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 BlockIDs (modeled simply via the naturals), as captured by the partial function definition $\mathsf{Mem} \triangleq \mathsf{BlockID} \xrightarrow{fin} \mathsf{Block}$. 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) or physical pointers (integers). 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. *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, integer values must be aligned to 4-bytes). 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. 5.

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. 5). A free operation will first convert a pointer (either physical or logical) into a logical block (via toPtr defined in Fig. 5) and check if the block can be freed (captured by the premise $m(b) = (true, pa, n, c)$ in the second rule of Fig. 5), then proceed to update the memory with the information that the block has been freed (the conclusion of this rule). Similarly, for memory accesses (omitted in Fig.

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

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

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

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

$$\text{range}(\text{blk}) \triangleq \textbf{if } \text{blk}.\mathit{live} \wedge \text{blk}.pa \neq \mathit{undef} \textbf{ then } [\text{blk}.pa,\ \text{blk}.pa + \text{blk}.sz - 1] \textbf{ else } \emptyset$$

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

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

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

$$v_1 \mathbin{\overline{\wedge}} v_2 \triangleq \textbf{match } v_1, v_2 \textbf{ with } \mid \_, \mathit{undef} \Rightarrow v_1 \mid \mathit{undef}, \_ \Rightarrow v_2 \mid \_, \_ \Rightarrow \textbf{if } v_1 = v_2 \textbf{ then } v_1 \textbf{ else } \textbf{NB}$$

$$\begin{aligned}\text{lift}_M(f)(v_1, v_2) \triangleq\ & \textbf{if } (v_1, v_2) \in \text{LogicalPtr} \times (\text{Int} \setminus \{0\}) \textbf{ then } f(v_1, \text{toPtr}_M(v_2)) \mathbin{\overline{\wedge}} f(\text{toInt}_M(v_1), v_2) \\ & \textbf{elif } (v_1, v_2) \in (\text{Int} \setminus \{0\}) \times \text{LogicalPtr} \textbf{ then } f(\text{toPtr}_M(v_1), v_2) \mathbin{\overline{\wedge}} f(v_1, \text{toInt}_M(v_2)) \\ & \textbf{else } f(v_1, v_2)\end{aligned}$$

$$\begin{aligned}[\![\text{psub}]\!]_M(v_1, v_2) \triangleq\ & \textbf{if } (v_1, v_2) \in \text{LogicalPtr} \times \text{Int} \textbf{ then } [\![\text{sub}]\!]_{\text{CompCert}}(\text{toInt}_M(v_1), v_2) \\ & \textbf{elif } (v_1, v_2) \in \text{Int} \times \text{LogicalPtr} \textbf{ then } [\![\text{sub}]\!]_{\text{CompCert}}(v_1, \text{toInt}_M(v_2)) \\ & \textbf{else } [\![\text{sub}]\!]_{\text{CompCert}}(v_1, v_2)\end{aligned}$$

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

Fig. 5. Selected rules for the semantics of Archmage.

5), 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 [Kang et al. 2015]. 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. 5, 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. 5).

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 §2, 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. 5), 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. 5. 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 binary operations.

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 intermediate 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. 5 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 3.1), 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. 5, is *guaranteed* to succeed when combining the results of concretizing a pointer and lifting an integer, for every binary operation.

Theorem 3.1. *For any binary operator* $\otimes$ *and values* v_1, v_2: $[\![\otimes]\!]_M\ (v_1, v_2) \neq \mathbf{NB}$

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.

4 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.

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 §3, 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 §2, 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: §4.1 explains technical details related to updating CompCert, §4.2 details steps towards mitigating the performance overhead caused by formally considering integer-pointer casts, and §4.3 explains the improvement on generated assembly.

CompCertCast is available as a fully proved Coq implementation.

4.1 Modifying CompCert to Support Integer-Pointer Casts

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).

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 simulation, 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.

4.1.1 Mixed Simulations and Memory Relations. To address these challenges, we modify CompCert to show behavioral refinement by extending the concept of *mixed simulations* [Song et al. 2019] instead of relying solely on forwards simulations. Intuitively, a mixed simulation holds if (*i*) a

$$\begin{array}{l}\textbf{Inductive}\ \mathsf{val\_intptr}_M : \mathsf{Val} \to \mathsf{Val} \to \mathbb{P}\ :=\\ \quad \mid \cdots\\ \quad \mid\ \mathsf{val\_intptr\_ptr\_int}\ :\\ \qquad \mathsf{toInt}_M\ (b, ofs) = i \to (i \neq \mathit{undef}) \ \to\ \mathsf{val\_intptr}_M\ (b, ofs)\ i\end{array}$$

Fig. 6. 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. 5).

forwards simulation holds and the target is locally deterministic [Song et al. 2019][4], 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 §2.2. CompCertCast thus defines an additional memory relation, in addition to the three existing 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. 6 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. 5).

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.

[4] '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.

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:

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

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 §2. 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 [Song et al. 2019] 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.

4.1.2 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 ensure compatibility 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 or remove 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.

4.1.3 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.

```
1 int foo(long *p, long *q){
2   int i = (int) p;
3   int c1 = i < q;
4   int c2 = p < q;
5   return c1 + c2;
6 }
```

```
1 int foo(long *p, long *q){
2   int i = (int) p;
3   int c1 = i < q;
4   int c2 = i < q;
5   return c1 + c2;
6 }
```

```
1 int foo(long *p, long *q){
2   int i = (int) p;
3   int c1 = i < q;
4   int c2 = c1;
5   return c1 + c2;
6 }
```

Fig. 7. An example application of common subexpression elimination, which may only take place after replacing p with i on line 4 through cast propagation.

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 §3.

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.

4.2 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.

4.2.1 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 §2.2. 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 §2.2, 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. 7 gives an example of this phenomenon, where applying cast propagation opens up an additional chance to perform common subexpression elimination (CSE).

In Fig. 7, 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 propagation, CSE cannot be applied as there are no common subexpressions—however, applying cast propagation yields the second code snippet in Fig.

```
1 foo(void *p){
2   l1 = p;
3   i = (int) p;
4   l2 = p;
5   l3 = malloc(8);
6
7   memcpy(&l3, &l1, 4);
8   memcpy(&l3+4, &l2+4, 4); // l3 = p
9 }
```

```
1 foo (void *p) {
2   l1 = p;
3   i = (int) p;
4   l2 = i;
5   l3 = malloc(8);
6
7   memcpy(&l3, &l1, 4);
8   memcpy(&l3+4, &l2+4, 4); // l3 = ?
9 }
```

Fig. 8. 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.

7, 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. 7.

One challenge that arises from applying cast propagation is that we must carefully define the semantics of load operations. Figure 8 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 for 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 [Kang et al. 2015], 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) [Cytron et al. 1991] 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 [Barthe et al. 2014] and implemented a new copy propagation algorithm that relies on SSA, which is observed to be more efficient than that of original CompCert.

4.2.2 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. 9, 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`.

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, 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.

```
1 int foo (void *p) {
2   long stk[42];
3
4   int i = stk[3];
5   *p = 42;
6   int j = stk[3];
7   return i + j;
8 }
```

Fig. 9. A small program that takes as argument a pointer p and writes to p.

To alleviate the aforementioned limitation, we implement an extra flag[5] 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 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.

4.3 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 $x = \mathrm{alloc}(8)$ will (*i*) generate a new logical pointer p for $\mathrm{alloc}(8)$, (*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 §4.1—the extended memory relations and event refinements, accounting for the fact that integers

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

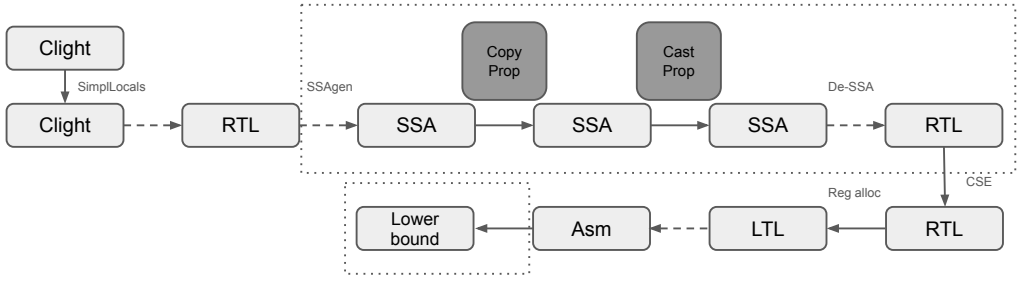

Fig. 10. The additional compilation passes applied by CompCertCast, indicated by the dotted box. Our new passes copy and cast propagation, and the lower bound improvement are highlighted.

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.[6]

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.

4.4 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. 10).

In the optimization chain (the upper dotted box), SSAgen and De-SSA are an application of static-single assignment from CompCert-SSA [Barthe et al. 2014]. Copy propagation is standard copy propagation; however, as CompCert did not perform copy propagation optimally as discussed in §4.2, we implemented a new, more efficient version from scratch. Implementing these three additional optimizations were prerequisites to implementing cast propagation as described in §4.2; 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 §4.3.

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

[6]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.

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 §4.2. We also tested that cast propagation was operating as intended on several small, hand-crafted programs. Interested readers may consult our Coq implementation [Kim et al. 2024].

5 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 [Xia et al. 2019] 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 [Song et al. 2023], 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 implementation. Existing verification frameworks are either incapable of supporting the complex integer-pointer casting patterns used in the xor-list [Lepigre et al. 2022], 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 [Reynolds 2002].

5.1 The Predicates and Rules of Archmage Logic

Archmage logic draws Iris-style separation logic built upon *resource algebras* (as introduced in Jung et al. [2018]) 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. 11, 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. 11. 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 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

User-Level Predicates and Relations

Block Data $\mathsf{m} = (b, sz) \in \mathsf{BlockID} \times \mathsf{Int}$

$$
\begin{array}{rcl}
p_1 \approx^{\mathsf{m}} p_2 & \triangleq & \exists ofs.\ \mathsf{offset}(\mathsf{m}, p_1, ofs) * \mathsf{offset}(\mathsf{m}, p_2, ofs) \\
\mathsf{live}^{\mathsf{m}}_q(p) & \triangleq & \mathsf{offset}(\mathsf{m}, p, 0) * \boxed{\mathsf{Allocated}_{q \in (0,1]}(\mathsf{m}.b)} \\
p \mapsto^{\mathsf{m}}_q v & \triangleq & \exists ofs.\ \mathsf{offset}(\mathsf{m}, p, ofs) * \boxed{\mathsf{Pointsto}_{q \in (0,1]}(\mathsf{m}.b, ofs, v)} \\
\mathsf{offset}(\mathsf{m}, p, ofs) & \triangleq & \boxed{BS(\mathsf{m}.b, \mathsf{m}.sz)} * (\ulcorner p = (\mathsf{m}.b, ofs) \urcorner \vee (\exists i.\ \boxed{BA(\mathsf{m}.b, i)} * \ulcorner p = i + ofs \urcorner)) \\
\mathsf{m}_1 \# \mathsf{m}_2 & \triangleq & \mathsf{m}_1.b \neq \mathsf{m}_2.b \qquad \mathsf{vld}(\mathsf{m}, ofs) \triangleq 0 \le ofs < \mathsf{m}.sz \qquad \mathsf{wvld}(\mathsf{m}, ofs) \triangleq 0 \le ofs \le \mathsf{m}.sz
\end{array}
$$

Selected Rules for Predicates and Relations

$$
\begin{array}{ll}
p_1 \approx^{\mathsf{m}} p_2 \twoheadrightarrow (p_1 \approx^{\mathsf{m}} p_2 * p_1 \approx^{\mathsf{m}} p_2) \quad (1) & \mathsf{live}^{\mathsf{m}}_{q_1+q_2}(p) \mathrel{-\!\!*\!\!-} (\mathsf{live}^{\mathsf{m}}_{q_1}(p) * \mathsf{live}^{\mathsf{m}}_{q_2}(p)) \quad (5) \\
p_1 \approx^{\mathsf{m}} p_2 \twoheadrightarrow (p_1 + k) \approx^{\mathsf{m}} (p_2 + k) \quad (2) & p \mapsto^{\mathsf{m}}_{q_1+q_2} v \mathrel{-\!\!*\!\!-} (p \mapsto^{\mathsf{m}}_{q_1} v * p \mapsto^{\mathsf{m}}_{q_2} v) \quad (6) \\
p_1 \approx^{\mathsf{m}} p_2 \twoheadrightarrow p_2 \approx^{\mathsf{m}} p_1 \quad (3) & p_1 \approx^{\mathsf{m}} p_2 \twoheadrightarrow (\mathsf{live}^{\mathsf{m}}_q(p_1) \mathrel{-\!\!*\!\!-} \mathsf{live}^{\mathsf{m}}_q(p_2)) \quad (7) \\
(p_1 \approx^{\mathsf{m}} p_2 * p_2 \approx^{\mathsf{m}} p_3) \twoheadrightarrow p_1 \approx^{\mathsf{m}} p_3 \quad (4) & p_1 \approx^{\mathsf{m}} p_2 \twoheadrightarrow (p_1 \mapsto^{\mathsf{m}}_q v \mathrel{-\!\!*\!\!-} p_2 \mapsto^{\mathsf{m}}_q v) \quad (8)
\end{array}
$$

$$
i_1 \approx^{\mathsf{m}} i_2 \twoheadrightarrow \ulcorner i_1 = i_2 \urcorner \ \text{ for integers } i_1, i_2 \qquad (9)
$$

$$
(\ulcorner \mathsf{vld}(\mathsf{m}_1, ofs_1) \wedge \mathsf{vld}(\mathsf{m}_2, ofs_2) \urcorner * \mathsf{live}^{\mathsf{m}_1}_{q_1}(p - ofs_1) * \mathsf{live}^{\mathsf{m}_2}_{q_2}(p - ofs_2)) \twoheadrightarrow \ulcorner \mathsf{m}_1 = \mathsf{m}_2 \wedge ofs_1 = ofs_2 \urcorner \qquad (10)
$$

Rules for Commands

$$
\begin{array}{l}
\{\ \ulcorner n > 0 \urcorner\ \}\ \mathsf{alloc}(n)\ \{\ r.\ \exists \mathsf{m}.\ \ulcorner \mathsf{m}.sz = n \urcorner * \mathsf{live}^{\mathsf{m}}_1(r) * (\circledast_{k \in [0, \mathsf{m}.sz)} (r + k) \mapsto^{\mathsf{m}}_1 \mathit{undef})\ \} \\
\{\ \mathsf{live}^{\mathsf{m}}_1(p) * (\circledast_{k \in [0, \mathsf{m}.sz)} (p + k) \mapsto^{\mathsf{m}}_1 \_)\ \}\ \mathsf{free}(p)\ \{\ \top\ \} \\
\{\ p \mapsto^{\mathsf{m}}_q v\ \}\ \ \mathsf{load}(p)\ \ \{\ r.\ \ulcorner r = v \urcorner * p \mapsto^{\mathsf{m}}_q v\ \} \qquad \{\ \mathsf{live}^{\mathsf{m}}_q(p - ofs)\ \}\ \mathsf{ptoi}(p)\ \{\ r.\ p \approx^{\mathsf{m}} r * \mathsf{live}^{\mathsf{m}}_q(p - ofs)\ \} \\
\{\ p \mapsto^{\mathsf{m}}_1 \_\ \}\ \mathsf{store}(p, v)\ \{\ p \mapsto^{\mathsf{m}}_1 v\ \} \qquad \{\ \top\ \}\ \mathsf{itop}(i)\ \{\ r.\ \ulcorner r = i \urcorner\ \} \\
\left\{ \begin{array}{l} \ulcorner \mathsf{wvld}(\mathsf{m}, ofs_1) \wedge \mathsf{wvld}(\mathsf{m}, ofs_2) \urcorner * \\ \mathsf{live}^{\mathsf{m}}_{q_1}(p_1 - ofs_1) * \mathsf{live}^{\mathsf{m}}_{q_2}(p_2 - ofs_2) \end{array} \right\}\ p_1 \otimes p_2\ \left\{ \begin{array}{l} r.\ \ulcorner r = ofs_1 \otimes ofs_2 \urcorner * \\ \mathsf{live}^{\mathsf{m}}_{q_1}(p_1 - ofs_1) * \mathsf{live}^{\mathsf{m}}_{q_2}(p_2 - ofs_2) \end{array} \right\} \\
\left\{ \begin{array}{l} \ulcorner \mathsf{m}_1 \# \mathsf{m}_2 \wedge \mathsf{vld}(\mathsf{m}_1, ofs_1) \wedge \mathsf{vld}(\mathsf{m}_2, ofs_2) \urcorner * \\ \mathsf{live}^{\mathsf{m}_1}_{q_1}(p_1 - ofs_1) * \mathsf{live}^{\mathsf{m}_2}_{q_2}(p_2 - ofs_2) \end{array} \right\}\ p_1 == p_2\ \left\{ \begin{array}{l} r.\ \ulcorner r = \mathit{false} \urcorner * \\ \mathsf{live}^{\mathsf{m}_1}_{q_1}(p_1 - ofs_1) * \mathsf{live}^{\mathsf{m}_2}_{q_2}(p_2 - ofs_2) \end{array} \right\}
\end{array}
$$

Fig. 11. User-level predicates, command rules, and selected rules for predicates in Archmage logic.

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*) $\mathsf{vld}(\mathsf{m}, ofs)$, checking that ofs is a valid with respect to block data m (*i.e.,* that a pointer $p = (\mathsf{m}.b, ofs)$ is an interior pointer), and (*iii*) $\mathsf{wvld}(\mathsf{m}, ofs)$, encoding the same idea for weak validity of pointers (*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. 11. 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 [Kim et al. 2024]): $BS(\mathsf{m}.b, \mathsf{m}.sz)$ 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 $\mathsf{Pointsto}_{q\in(0,1]}(\mathsf{m}.b, ofs, v)$, 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. 11, 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. 11, 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) $\mathsf{live}^{\mathsf{m}}_1(r)$, *i.e.,* that r is a live pointer pointing to the head of the newly allocated block m, and (ii) $(\divideontimes_{k\in[0,\mathsf{m}.sz)}(r+k) \mapsto^{\mathsf{m}}_1 undef)$, *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*.

```
1  struct node {
2    long item;  // value in node
3    long link;  // xor prev next
4  }
5  long delete_hd(node** hdH, node** tlH) {
6    long item = 0; // empty list
7    node* hd_old = *hdH;
8
9    if (hd_old != NULL) { // non-empty list
10     item = hd_old->item;
11     node *hd_new = (node*) hd_old->link;
12     *hdH = hd_new;
13     if (hd_new == NULL) {
14       *tlH = NULL;
15     } else {
16       intptr_t link = hd_new->link;
17       hd_new->link = link ^ (intptr_t)hd_old;
18     }
19     free(hd_old);
20   }
21   return item;
22 }
23 long delete_tl(node** hdH, node** tlH)
24 ...
```

Fig. 12. Snippets of an xor-list implementation, showing the code for struct node and the function delete_hd.

5.2 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 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 [Lepigre et al. 2022] fails on the xor-list because they cannot resolve which block xor prv (xor prv nxt) should point to.[7]

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. 12. 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 [Kim et al. 2024]; we 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. 13. The specification is simple on a high level: given an xor-list

[7]The Quasi-Concrete model [Kang et al. 2015]. or PNVI-ae-udi [Memarian et al. 2019], can support xor-lists, but do not come with a program logic.

Xorlist Spec

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

Pre & Post Conditions

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

Fig. 13. Correctness specifications for the functions add_hd, add_tl, delete_hd, and delete_hd.

of the form x :: xs (encoded by the precondition $\mathtt{xorlist}_1$ hdH tlH (x :: xs)), delete_hd simply returns the removed value x and guarantees that x is removed from the xor-list.[8]

The definition of the predicate xorlist is given on the top of Fig. 13: 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 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 ($\mathtt{frag}_q$ $\mathsf{m}_{\mathtt{NULL}}$ $\mathsf{m}_{\mathtt{NULL}}$ NULL hd tl NULL xs). The predicate frag is defined on the top of Fig. 13, and encodes the actual xor-list traversal methodology described at the start of this section—observe the case in which xs = x :: xs′, in which frag requires that hd (*i.e.,* the current head) points to a valid node[9] 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.,* $\mathtt{frag}_q$ 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. 13. Fig. 14 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. 14 (❶ to ⓬). Changes in a postcondition compared to a precondition are highlighted in red.

[8]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.

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

❶ $\{\ \mathsf{xorlist}_1\ \mathsf{hdH}\ \mathsf{tlH}\ (\mathsf{x} :: \mathsf{xs})\ \}$

```
long delete_hd(node** hdH, node** tlH) {
```

❷ $\left\{\begin{array}{l}\mathsf{hdH} \mapsto_1^{\mathsf{m_{hdH}}} \mathsf{hd} * \mathsf{tlH} \mapsto_1^{\mathsf{m_{tlH}}} \mathsf{tl} * \mathsf{m_{hd}}.sz = 2 * \mathsf{live}_1^{\mathsf{m_{hd}}}(\mathsf{hd}) * \mathsf{hd} \mapsto_1^{\mathsf{m_{hd}}} [\mathsf{x}, i_{\mathsf{mid}}] * \\ \qquad \mathsf{frag}_1\ \mathsf{m_{hd}}\ \mathsf{m_{NULL}}\ \mathsf{hd}\ i_{\mathsf{mid}}\ \mathsf{tl}\ \mathsf{NULL}\ \mathsf{xs}\end{array}\right\}$

```
  long item = 0;
  node* hd_old = *hdH;
  if (hd_old != NULL) {
    item = hd_old->item;
    node *hd_new = (node*) hd_old->link;
```

❸ $\left\{\begin{array}{l}\mathsf{hdH} \mapsto_1^{\mathsf{m_{hdH}}} \mathsf{hd} * \mathsf{tlH} \mapsto_1^{\mathsf{m_{tlH}}} \mathsf{tl} * \mathsf{m_{hd}}.sz = 2 * \mathsf{live}_1^{\mathsf{m_{hd}}}(\mathsf{hd}) * \mathsf{hd} \mapsto_1^{\mathsf{m_{hd}}} [\mathsf{x}, i_{\mathsf{mid}}] * \\ \mathsf{frag}_1\ \mathsf{m_{hd}}\ \mathsf{m_{NULL}}\ \mathsf{hd}\ i_{\mathsf{mid}}\ \mathsf{tl}\ \mathsf{NULL}\ \mathsf{xs} * \mathsf{hd\_old} = \mathsf{hd} * \mathsf{item} = \mathsf{x} * \mathsf{hd\_new} = i_{\mathsf{mid}}\end{array}\right\}$

```
    *hdH = hd_new;
```

❹ $\left\{\begin{array}{l}\mathsf{hdH} \mapsto_1^{\mathsf{m_{hdH}}} i_{\mathsf{mid}} * \mathsf{tlH} \mapsto_1^{\mathsf{m_{tlH}}} \mathsf{tl} * \mathsf{m_{hd}}.sz = 2 * \mathsf{live}_1^{\mathsf{m_{hd}}}(\mathsf{hd}) * \mathsf{hd} \mapsto_1^{\mathsf{m_{hd}}} [\mathsf{x}, i_{\mathsf{mid}}] * \\ \mathsf{frag}_1\ \mathsf{m_{hd}}\ \mathsf{m_{NULL}}\ \mathsf{hd}\ i_{\mathsf{mid}}\ \mathsf{tl}\ \mathsf{NULL}\ \mathsf{xs} * \mathsf{hd\_old} = \mathsf{hd} * \mathsf{item} = \mathsf{x} * \mathsf{hd\_new} = i_{\mathsf{mid}}\end{array}\right\}$

```
    if (hd_new == NULL) {
```

❺ $\left\{\begin{array}{l}\mathsf{hdH} \mapsto_1^{\mathsf{m_{hdH}}} \mathsf{NULL} * \mathsf{tlH} \mapsto_1^{\mathsf{m_{tlH}}} \mathsf{tl} * \mathsf{m_{hd}}.sz = 2 * \mathsf{live}_1^{\mathsf{m_{hd}}}(\mathsf{hd}) * \mathsf{hd} \mapsto_1^{\mathsf{m_{hd}}} [\mathsf{x}, \mathsf{NULL}] * \\ \mathsf{frag}_1\ \mathsf{m_{hd}}\ \mathsf{m_{NULL}}\ \mathsf{hd}\ \mathsf{NULL}\ \mathsf{tl}\ \mathsf{NULL}\ \mathsf{xs} * \mathsf{hd\_old} = \mathsf{hd} * \mathsf{item} = \mathsf{x}\end{array}\right\}$

```
      *tlH = NULL;
```

❻ $\left\{\begin{array}{l}\mathsf{hdH} \mapsto_1^{\mathsf{m_{hdH}}} \mathsf{NULL} * \mathsf{tlH} \mapsto_1^{\mathsf{m_{tlH}}} \mathsf{NULL} * \mathsf{m_{hd}}.sz = 2 * \mathsf{live}_1^{\mathsf{m_{hd}}}(\mathsf{hd}) * \mathsf{hd} \mapsto_1^{\mathsf{m_{hd}}} [\mathsf{x}, \mathsf{NULL}] * \\ \mathsf{frag}_1\ \mathsf{m_{hd}}\ \mathsf{m_{NULL}}\ \mathsf{hd}\ \mathsf{NULL}\ \mathsf{tl}\ \mathsf{NULL}\ \mathsf{xs} * \mathsf{hd\_old} = \mathsf{hd} * \mathsf{item} = \mathsf{x}\end{array}\right\}$

```
    } else {
```

❼ $\left\{\begin{array}{l}\mathsf{hdH} \mapsto_1^{\mathsf{m_{hdH}}} i_{\mathsf{mid}} * \mathsf{tlH} \mapsto_1^{\mathsf{m_{tlH}}} \mathsf{tl} * \mathsf{m_{hd}}.sz = 2 * \mathsf{live}_1^{\mathsf{m_{hd}}}(\mathsf{hd}) * \mathsf{hd} \mapsto_1^{\mathsf{m_{hd}}} [\mathsf{x}, i_{\mathsf{mid}}] * \\ \mathsf{xs} = \mathsf{x}' :: \mathsf{xs}' * \mathsf{m_{mid}}.sz = 2 * \mathsf{hd} \approx^{\mathsf{m_{hd}}} i_{\mathsf{hd}} * \mathsf{live}_1^{\mathsf{m_{mid}}}(i_{\mathsf{mid}}) * i_{\mathsf{mid}} \mapsto_1^{\mathsf{m_{mid}}} [\mathsf{x}', \mathsf{xor}\ i_{\mathsf{hd}}\ i_{\mathsf{mid}'}] * \\ \mathsf{frag}_1\ \mathsf{m_{mid}}\ \mathsf{m_{NULL}}\ i_{\mathsf{mid}}\ i_{\mathsf{mid}'}\ \mathsf{tl}\ \mathsf{NULL}\ \mathsf{xs}' * \\ \mathsf{hd\_old} = \mathsf{hd} * \mathsf{item} = \mathsf{x} * \mathsf{hd\_new} = i_{\mathsf{mid}}\end{array}\right\}$

```
      intptr_t link = hd_new->link;
```

❽ $\left\{\begin{array}{l}\mathsf{hdH} \mapsto_1^{\mathsf{m_{hdH}}} i_{\mathsf{mid}} * \mathsf{tlH} \mapsto_1^{\mathsf{m_{tlH}}} \mathsf{tl} * \mathsf{m_{hd}}.sz = 2 * \mathsf{live}_1^{\mathsf{m_{hd}}}(\mathsf{hd}) * \mathsf{hd} \mapsto_1^{\mathsf{m_{hd}}} [\mathsf{x}, i_{\mathsf{mid}}] * \\ \mathsf{xs} = \mathsf{x}' :: \mathsf{xs}' * \mathsf{m_{mid}}.sz = 2 * \mathsf{hd} \approx^{\mathsf{m_{hd}}} i_{\mathsf{hd}} * \mathsf{live}_1^{\mathsf{m_{mid}}}(i_{\mathsf{mid}}) * i_{\mathsf{mid}} \mapsto_1^{\mathsf{m_{mid}}} [\mathsf{x}', \mathsf{xor}\ i_{\mathsf{hd}}\ i_{\mathsf{mid}'}] * \\ \mathsf{frag}_1\ \mathsf{m_{mid}}\ \mathsf{m_{NULL}}\ i_{\mathsf{mid}}\ i_{\mathsf{mid}'}\ \mathsf{tl}\ \mathsf{NULL}\ \mathsf{xs}' * \\ \mathsf{hd\_old} = \mathsf{hd} * \mathsf{item} = \mathsf{x} * \mathsf{hd\_new} = i_{\mathsf{mid}} * \mathsf{link} = \mathsf{xor}\ i_{\mathsf{hd}}\ i_{\mathsf{mid}'}\end{array}\right\}$

```
      hd_new->link= link ^ (intptr_t)hd_old;
```

❾ $\left\{\begin{array}{l}\mathsf{hdH} \mapsto_1^{\mathsf{m_{hdH}}} i_{\mathsf{mid}} * \mathsf{tlH} \mapsto_1^{\mathsf{m_{tlH}}} \mathsf{tl} * \mathsf{m_{hd}}.sz = 2 * \mathsf{live}_1^{\mathsf{m_{hd}}}(\mathsf{hd}) * \mathsf{hd} \mapsto_1^{\mathsf{m_{hd}}} [\mathsf{x}, i_{\mathsf{mid}}] * \\ \mathsf{xs} = \mathsf{x}' :: \mathsf{xs}' * \mathsf{m_{mid}}.sz = 2 * \mathsf{hd} \approx^{\mathsf{m_{hd}}} i_{\mathsf{hd}} * \mathsf{live}_1^{\mathsf{m_{mid}}}(i_{\mathsf{mid}}) * i_{\mathsf{mid}} \mapsto_1^{\mathsf{m_{mid}}} [\mathsf{x}', i_{\mathsf{mid}'}] * \\ \mathsf{frag}_1\ \mathsf{m_{mid}}\ \mathsf{m_{NULL}}\ i_{\mathsf{mid}}\ i_{\mathsf{mid}'}\ \mathsf{tl}\ \mathsf{NULL}\ \mathsf{xs}' * \\ \mathsf{hd\_old} = \mathsf{hd} * \mathsf{item} = \mathsf{x} * \mathsf{hd\_new} = i_{\mathsf{mid}} * \mathsf{link} = \mathsf{xor}\ i_{\mathsf{hd}}\ i_{\mathsf{mid}'}\end{array}\right\}$

```
    }
```

❿ $\left\{\begin{array}{l}\mathsf{hd\_old} = \mathsf{hd} * \mathsf{m_{hd}}.sz = 2 * \mathsf{live}_1^{\mathsf{m_{hd}}}(\mathsf{hd}) * \mathsf{hd} \mapsto_1^{\mathsf{m_{hd}}} [\_, \_] * \\ \mathsf{item} = \mathsf{x} * \exists\, \mathsf{m_{hdH}}\ \mathsf{m_{tlH}}\ i_{\mathsf{mid}}\ \mathsf{tl},\ \mathsf{hdH} \mapsto_1^{\mathsf{m_{hdH}}} i_{\mathsf{mid}} * \mathsf{tlH} \mapsto_1^{\mathsf{m_{tlH}}} \mathsf{tl} * \mathsf{frag}_1\ \mathsf{m_{NULL}}\ \mathsf{m_{NULL}}\ \mathsf{NULL}\ i_{\mathsf{mid}}\ \mathsf{tl}\ \mathsf{NULL}\ \mathsf{xs}\end{array}\right\}$

```
    free(hd_old);
```

⓫ $\{\mathsf{item} = \mathsf{x} * \exists\, \mathsf{m_{hdH}}\ \mathsf{m_{tlH}}\ i_{\mathsf{mid}}\ \mathsf{tl},\ \mathsf{hdH} \mapsto_1^{\mathsf{m_{hdH}}} i_{\mathsf{mid}} * \mathsf{tlH} \mapsto_1^{\mathsf{m_{tlH}}} \mathsf{tl} * \mathsf{frag}_1\ \mathsf{m_{NULL}}\ \mathsf{m_{NULL}}\ \mathsf{NULL}\ i_{\mathsf{mid}}\ \mathsf{tl}\ \mathsf{NULL}\ \mathsf{xs}\}$

```
  }
  return item;
}
```

⓬ $\{\ r.\ \ r = \mathsf{x} * \mathsf{xorlist}_1\ \mathsf{hdH}\ \mathsf{tlH}\ \mathsf{xs}\ \}$

Fig. 14. 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.

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 hd_old ≠ NULL follows from hd_old = hd and $\mathsf{hd} \mapsto_1^{m_{\mathsf{hd}}} [\mathsf{x}, i_{\mathsf{mid}}]$.

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 hd_new = i_{mid} to obtain that $\mathsf{hdH} \mapsto_1^{m_{\mathsf{hdH}}} i_{\mathsf{mid}}$.

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 hd_new = i_{mid} = 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 $\mathtt{frag}_1\ m_{\mathsf{hd}}\ m_{\mathsf{NULL}}\ \mathsf{hd}\ i_{\mathsf{mid}}\ \mathsf{tl}\ \mathsf{NULL}\ \mathsf{xs}$ 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_{mid} = hd_new ≠ 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 hd_new = i_{mid} and $i_{\mathsf{mid}} \mapsto_1^{m_{\mathsf{mid}}} [\mathsf{x}', \mathsf{xor}\ i_{\mathsf{hd}}\ i_{\mathsf{mid}'}]$ to obtain link = xor i_{hd} $i_{\mathsf{mid}'}$.

❽ to ❾ is a store operation that encodes the new `link` for the new head hd_new: because hd_old = hd $\approx^{m_{\mathsf{hd}}} i_{\mathsf{hd}}$ and link = xor i_{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 ⟹ ❿ and ❾ ⟹ ❿; for ❻ ⟹ ❿, i_{mid} = tl = NULL in ❿ serves as witness for the implication to hold. In the case of ❾ ⟹ ❿, the implication holds as the red $\mathtt{frag}_1\ m_{\mathsf{NULL}}\ m_{\mathsf{NULL}}\ \mathsf{NULL}\ i_{\mathsf{mid}}\ \mathsf{tl}\ \mathsf{NULL}\ \mathsf{xs}$ in ❿ is the result of re-folding the predicates about i_{mid} on the second and third line of ❾, while hd $\approx^{m_{\mathsf{hd}}} i_{\mathsf{hd}}$, hd_new = i_{mid}, link = xor i_{hd} $i_{\mathsf{mid}'}$ and xs = x′ :: 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 5.1.

Theorem 5.1 (XorlistReverse). *For any value* q, v_1, v_2, xs*:*

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

Theorem 5.1 allows us to reuse the specifications listed in $\mathtt{frag}_q$ and $\mathtt{xorlist}_q$, from the top of Fig. 13, to prove the correctness of `delete_tl` as well. Without Theorem 5.1 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.

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.

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. 15 depicts a small function `main` that uses the xor-list; verifying `main` took around an additional 250 LoC in total.

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.

```
1 #include "xorlist.h"
2 int main() {
3   node *head = NULL, *tail = NULL;
4   long item = 1;
5   add_hd(&head, &tail, 3);
6   add_tl(&head, &tail, 7);
7   item = item * delete_hd(&head, &tail);
8   item = item * delete_tl(&head, &tail);
9   return item;
10 }
```

Fig. 15. A code fragment illustrating a simple function `main` that is a client of the xor-list defined in `xorlist.h`.

6 Discussion and Related Work

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 §4.2); 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 [Besson et al. 2014, 2015, 2019] 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 [Kang et al. 2015]. 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 §5, 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 [Besson et al. 2019]. 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 function inlining, that would require additional work on the symbolic model to support in CompCertS.

Other work on formalizing integer-pointer casts. Stackaware CompCert [Wang et al. 2019] 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 §4.3 thus represents an orthogonal improvement.

VST [Appel 2014; Mansky and Du 2024] 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 [Appel et al. 2023], and thus VST is incapable of supporting integer-pointer casts as well.

The Quasi-Concrete model [Kang et al. 2015] 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 §4, 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 [Sammler et al. 2023] 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 [Memarian et al. 2019] 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 [Lepigre et al. 2022] 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 [Sammler et al. 2021], 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 [Lee et al. 2018] 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 §3.

seL4 [Klein et al. 2009] 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.

7 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 §5.2. We hope to extend the work presented in this paper by automating Archmage logic, to obtain a full verification chain capable of automatically verifying and compiling integer-pointer casting programs.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. Chung-Kil Hur is the corresponding author. This research was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT2102-03.

Data-Availability Statement

The Coq formalization of this paper may be found at [Kim et al. 2024].

References

Andrew W. Appel. 2014. *Program Logics for Certified Compilers.* Cambridge University Press. https://doi.org/10.1017/CBO9781107256552

Andrew W. Appel, Lennart Beringer, Qinxiang Cao, and Josiah Dodds. 2023. *Verifiable C.* Vol. 1. https://raw.githubusercontent.com/PrincetonUniversity/VST/master/doc/VC.pdf

Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 4 (mar 2014), 35 pages. https://doi.org/10.1145/2579080

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2014. A Precise and Abstract Memory Model for C Using Symbolic Values. In *Asian Symposium on Programming Languages and Systems.* https://doi.org/10.1007/978-3-319-12736-1_24

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2015. A Concrete Memory Model for CompCert. In *6th International Conference on Interactive Theorem Proving (ITP 2015).* https://doi.org/10.1007/978-3-319-22102-1_5

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2019. CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics. In *Journal of Automated Reasoning.* https://doi.org/10.1007/s10817-018-9496-y

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05).* Association for Computing Machinery, New York, NY, USA, 259–270. https://doi.org/10.1145/1040305.1040327

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–72. https://doi.org/10.1007/3-540-44898-5_4

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451490. https://doi.org/10.1145/115372.115320

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

Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. 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).* https://doi.org/10.1145/2737924.2738005

Yonghyun Kim, Minki Cho, Jaehyung Lee, Jinwoo Kim, Taeyoung Yoon, Youngju Song, and Chung-Kil Hur. 2024. Artifact: Archmage and CompCertCast: End-to-End Verification Supporting Integer-Pointer Casting *(POPL 2025 Artifact).* https://doi.org/10.5281/zenodo.13939050

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal verification of an OS kernel. In *SOSP.* ACM, 207–220. https://doi.org/10.1145/1629575.1629596

Chris Lattner and Vikram Adve. 2004. 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* (Palo Alto, California) *(CGO '04).* IEEE Computer Society, USA, 75. https://doi.org/10.1109/CGO.2004.1281665

Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling High-Level Optimizations and Low-Level Code in LLVM. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 125 (oct 2018), 28 pages. https://doi.org/10.1145/3276495

Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. 2022. VIP: Verifying Real-World C Idioms with Integer-Pointer Casts. *Proc. ACM Program. Lang.* 6, POPL, Article 20 (jan 2022), 32 pages. https://doi.org/10.1145/3498681

Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006).*

William Mansky and Ke Du. 2024. An Iris Instance for Verifying CompCert C Programs. *Proc. ACM Program. Lang.* 8, POPL, Article 6 (jan 2024), 27 pages. https://doi.org/10.1145/3632848

Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.* 3, POPL, Article 67 (jan 2019), 32 pages. https://doi.org/10.1145/3290380

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02).* IEEE Computer Society, USA, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021).*

Association for Computing Machinery, New York, NY, USA, 158–174. https://doi.org/10.1145/3453483.3454036

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-Language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 27 (jan 2023), 31 pages. https://doi.org/10.1145/3571220

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2019. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371091

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (jan 2023), 31 pages. https://doi.org/10.1145/3571232

The Coq Development Team. 2021. The Coq Proof Assistant 8.13.2 Reference Manual. https://coq.github.io/doc/V8.13.2/refman/.

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (jan 2019), 30 pages. https://doi.org/10.1145/3290375

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371119

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