End-to-end Compositional Verification of Program Safety through Verified and Verifying Compilation

JINHUA WU, Shanghai Jiao Tong University, China
YUTING WANG*, Shanghai Jiao Tong University, China
LIUKUN YU, Shanghai Jiao Tong University, China
LINGLONG MENG, University of Minnesota, USA

Safety properties (e.g., absence of undefined behaviors) are critical for correct operation of programs. Those properties are usually verified at the source level (e.g., by separation logics) and preserved to the target by verified compilers (e.g., CompCert), thereby achieving end-to-end verification of safety. However, modern safe programming languages like Rust pose new problems in achieving end-to-end safety. Because not all functionalities can be implemented in the safe language, mixing safe and unsafe modules is needed. Therefore, verified compilation must preserve a modular notion of safety which can be composed at the target level. Furthermore, certain classes of errors (e.g., memory errors) are automatically excluded by verifying compilation (e.g., borrow checking) for modules written in safe languages. As a result, verified compilation needs to cooperate with verifying compilation to ensure end-to-end safety.

To address the above problems, we propose a language-agnostic approach to end-to-end compositional verification of program safety. Our approach provides a modular and generic definition of safety called *open safety* based on program semantics described as open labeled transition systems (LTS). By exploiting the compatible interfaces of open LTS, open safety is composable at the boundary of modules. Furthermore, it can be modularly preserved by verified compilers supporting compositional verification. Those properties enable separate verification of safety for heterogeneous modules and composition of the safety results at the target level. Open safety can be generalized to *partial* safety (i.e., only a certain class of errors can occur). By this we formalized the correctness of verifying compilation as derivation of total safety from partial safety. We demonstrate how our framework can combine verified and verifying compilation by developing a verified compiler for an ownership language (called Owlang) inspired by Rust. We evaluate our approach on the compositional safety verification using a hash map implemented by Owlang and C.

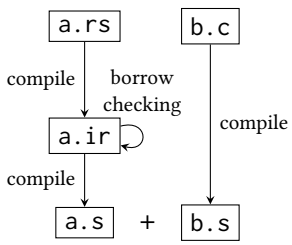
1 Introduction

An ultimate goal of formal verification is to ensure executable code running on hardware or OS does not go wrong because of program errors (e.g., memory error, division by zero), i.e., the executable code is *safe*. Instead of directly targeting executable code, a large number of safety verification techniques (e.g., separation logic [40], type systems [15]) work on source programs as they are easier to reason about with richer structures. In the end, the gap between safety guarantees for source and target could be bridged by a verified compiler which propagates source-level safety properties down to target code. A typical example is the Verified Software Toolchain (VST) [4, 6], which is a higher-order separation logic for proving safety of C code and relies on CompCert[28]—the state-of-the-art verified C compiler—to propagate this safety down to assembly code.

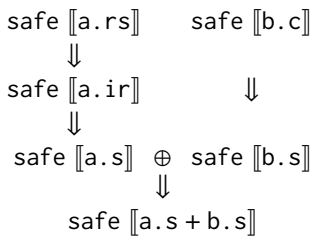
As software industry has been long plagued by security attacks caused by memory errors (e.g., invalid pointers, use-after-free) [10], there is an increasing trend to steer away from programming languages inherently not memory safe like C and C++ [18], and to embrace languages like Rust with memory safety guaranteed by safety checking of its compiler (e.g., borrow checking) [49].

*Corresponding author

Authors' Contact Information: Jinhua Wu, John Hopcroft Center for Computer Science, School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, China, jinhua.wu@sjtu.edu.cn; Yuting Wang, John Hopcroft Center for Computer Science, School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, China, yuting.wang@sjtu.edu.cn; Liukun Yu, School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, China, yu_liukun@outlook.com; Linglong Meng, University of Minnesota, USA, meng0181@umn.edu.



(a) Compilation and Linking



(b) Safety Preservation and Composition

Fig. 1. Verifying Safety for a Program Mixing Rust and C Code

Still, programming tasks demanding manipulation of low-level memory (e.g., pointer arithmetic) cannot be implemented in safe Rust because they are rejected by borrow checking. Instead, they need to be implemented in the unsafe fragment of Rust, or in unsafe languages like C. As a result, modern software is increasingly composed of modules written in both safe and unsafe languages. A prominent example is the Linux kernel which now contains both C and Rust components [43].

In this paper, we are concerned with verifying the safety of target code compiled from source programs mixing safe and unsafe code. Fig. 1 depicts a typical example. The module `a.rs` is implemented in safe Rust. It is compiled to assembly code `a.s` going through borrow checking in the MIR intermediate language. The module `b.c` is implemented in C because it needs to manipulate low-level memory. By encapsulating `b.c` under a safe interface, `b.c` could be invoked by `a.rs` like a regular Rust module. After the source modules are compiled to assembly, they are linked into a complete program `a.s + b.s`. Our goal is to verify the safety of `a.s + b.s`. Following the approach of VST described in the beginning paragraph, an obvious plan is depicted in Fig. 1b. First, we individually verify the safety of source code `a.rs` and `b.c` (e.g., by exploiting separation logics for Rust and C). We then utilize verified compilers to propagate safety of `a.rs` and `b.c` down to assembly level. Finally, if the safety assumptions between `a.rs` and `b.c` are correct, we should be able to prove that the target code `a.s + b.s` is safe from the individual safety of `a.s` and `b.s`.

If the above plan works, we get an approach to *end-to-end* verification of safety guarantees *compositionally*, i.e., verification is carried out separately in source modules, and compiled and composed to form safety of the final target. This approach could greatly reduce the efforts for verifying safety for heterogeneous programs by exploiting off-the-shelf verification tools for different languages (e.g. VST and RustBelt [22]) and verified optimizing compilers (e.g., CompCert).

Despite the aforementioned potential, it is not obvious at all how the above approach could be concretely developed. In particular, it is not even clear what the definition of safety in Fig. 1b could be for it to be preserved by realistic optimizing compilers. Below we analyze existing approaches to safety verification to bring out the problems.

1.1 Problems

1.1.1 Preservation of Safety by Verified Compilation of Heterogeneous Modules. A major goal of the above approach is to support preservation of safety for modules written in various safe and unsafe languages by verified compilers. On the surface, this is easy to achieve by using existing safety definitions in source-level verification tools, and by exploiting semantics-preserving compilers supporting *verified compositional compilation* (VCC) [13, 21, 24, 45, 47, 53, 55], i.e., separate compilation of heterogeneous modules and composition of compilation correctness. However, it turns out that either obvious safety definitions are not compatible with VCC, or they require a

daunting amount of changes to verified compilers which defeat our goal of using compilers *as they are*. Below, we discuss the problems of combining common safety verification techniques, i.e., separation logics and type systems, with the state-of-the-art of VCC [24, 55], i.e., extensions of CompCert based on simulation of open modules originating from Compositional CompCert [47].

With separation logics, the standard safety definition is *partial correctness* in Hoare logics [16]. A module M is partially correct, if there exists a Hoare triple $\{P\}M\{Q\}$, indicating that if M start with a state satisfying the precondition P, then it can always take the next step (i.e., does not crash or go wrong) as long as its execution has not reach an end. Furthermore, if M terminates then it must terminate in a state satisfying Q.

A Hoare triple $\{P\}M\{Q\}$ is a property about the module M *as a whole*. As such, semantics-preserving compilers for *complete* programs (e.g., the original CompCert) preserves partial correctness. For example, VST proves that if a C program M is partially correct, then the target code M' compiled from M by CompCert is also partially correct.

However, partial correctness cannot be preserved by the state-of-the-art compilers supporting VCC for *open modules*, i.e., modules that may invoke other modules. To see that, consider a Hoare triple $\{P\}M\{Q\}$ for the open module M. It asserts that every state S reachable from the initial state of M must be safe. Preserving this triple demands preservation of execution traces from initial states of M to S by compilation. Suppose such an execution trace include invocation of a module in the context of M, then it cannot be preserved by compiler correctness theorems for VCC, as they do not make any assumption about execution of modules living in the environment. Therefore, it is impossible to propagate partial correctness of open modules from source to target via VCC (a more fundamental explanation is that there are inherent conflicts between the "big-step" nature of partial correctness and "small-step" simulations used in VCC, which we will discuss in §2). As an evidence, VST defines a variant of partial correctness named safeN [6]. However, it is only used at C level and is not preserved by CompCert. As a result, partial correctness cannot be applied to our example in Fig. 1 where `a.rs` invokes unsafe implementation in `b.c` through its interface.

Besides separation logics, another class of mainstream technique for verifying safety is type systems [15]. A well-defined type system ensures *1)* every well-typed program cannot get stuck or go wrong in execution, also known as the *progress property*, and *2)* execution does not break well-typedness of programs. Then, a program P is safe if it is well-typed. Since types can be given to sub-components of complete programs—including modules, functions, expressions, etc.—safety of open modules can be easily defined and preserved by type-preserving compilers. In the end, safe modules can be composed to form complete programs via typing rules at the target level.

However, adopting well-typedness as safety in Fig. 1b comes with a high cost. Although there is prior work on *type-preserving compilation* [27, 33], integrating those techniques in semantics-preserving compilers require developing type systems for every intermediate language and proving that types of programs are preserved by every compiler pass. For example, CompCert has 9 intermediate languages with little type information and 20 passes. Making it type-preserving would be a project for multiple years.

In summary, we face the following question: **What is a modular definition of safety that is sufficiently strong so that it can be preserved by verified compilation, and sufficiently lightweight so that it does not require intrusive changes to existing verified compilers?**

1.1.2 Combining Safety Checking by Verifying Compilation with Verified Compilation. Modern systems programming languages relies on a combination of language features and safety checking in compilers to discharge safety errors. For example, to rule out memory errors, Rust introduces complex ownership and borrowing mechanisms so that any Rust program that can go through

borrow checking is (supposedly) memory safe. In other words, certain classes of safety properties do not need to be verified at the source-level, they are automatically established by compilers.

The idea of using compilers to discharge program errors and to formally prove their absence is known as *verifying compilation* [17]. The most prominent techniques of verifying compilation originates from proof-carrying code (PCC) [37]. In the initial version of PCC, the compiler produces target code along with a set of verification conditions (VC) [36]. Upon successfully discharging VC, the target code is guaranteed to be safe. Later, foundational proof-carrying code proposes a semantic type system such that any well-typed term is safe [5]. Hence verifying compilation can be implemented as type checking. Recent advances along this line include semantic type systems supporting multiple languages [39, 51].

However, the existing techniques for verifying compilation are not directly applicable to our example in Fig. 1. In our example, borrow checking can only guarantee the checked program `a.ir` is *partially* safe. That is, it only ensures that there is no memory error in `a.ir`. However, there can still be other errors such as division by zero in `a.ir`. We must rely on additional verification to discharge the remaining errors, and rely on verified compilation to propagate safety to `a.s`. On the other hand, existing verifying compilation techniques build systems to check that programs are *totally* safe, and they usually prove this total safety directly for target code without cooperating with verified compilers.

In summary, we face the following questions: **How could we describe the partial safety of a program, i.e., it is absent of a particular class of error and may contain other errors? Furthermore, how does verifying compilation for this partial safety cooperate with verified compilation to ensure the safety of final target program?**

1.2 Our Contributions

We answer the above questions by introducing a safety definition called *open safety* which supports uniform description of partial and total safety for open modules, and cooperation between verified and verifying compilation. With open safety, we give a concrete implementation of the approach to end-to-end and compositional verification of safety properties described in Fig. 1 based on the state-of-the-art verified compiler support VCC, i.e., CompCertO [24, 54, 55]. Our technical contributions are summarized as follows:

- We define open safety as a "small-step" variant of partial correctness on top of small-step operational semantics described as open labeled transition systems (LTS). It allows pre- and post-conditions on the boundary of modules implemented in arbitrary languages, thereby enabling compositional safety properties for heterogeneous modules. Note that open safety is sound (adequate) w.r.t. partial correctness, in that it implies partial correctness when applied to complete programs. By building in a state invariant for LTS, open safety can be modularly preserved by verified compilation by proving preservation of invariants under simulations. Moreover, because state invariants are *existentially quantified*, i.e., they are not visible outside of individual modules, proving preservation of open safety does not require any changes to the compilers or their correctness proofs. The above features make open safety a satisfying answer to the question at the end of §1.1.1. We demonstrate its effectiveness in this regard by proving that the entire compilation chain of CompCertO preserves open safety, from C modules all the way down to assembly code.
- We generalize open safety to be parameterized over a class of safety errors to describe *partial safety*, i.e., a module is partially safe if it may only contain those errors. The generalized open safety enjoys all the previously mentioned benefits for verified compilation. Moreover, in the same framework, it gives a formal treatment of verifying compilation, i.e, verifying

compilation is formalized as derivation of total safety from partial safety. This gives a satisfying answer to the question at the end of §1.1.2.

- To show that open safety can be checked and preserved by compilers combining verified and verifying compilation, we develop a compiler for a language called Owlang (*Ownership Language*) which can be viewed as a subset of core Rust. Like Rust, Owlang supports automatic heap management and ownership transfer of heap values. However, it does not support references or borrowing. We implement a compiler from Owlang to assembly by combining a frontend from Owlang to Clight with CompCert. In particular, the frontend contains a verifying pass called *ownership checking* (which is a fragment of the Rust borrow checker [42]) for ensuring memory safety under ownership transfer of heap values. We formally proved *1)* the semantics preservation of the *full* Owlang compiler, *2)* open safety is end-to-end preserved by the compiler, and *3)* the ownership checking pass is correct, i.e., there can be no *temporal* memory error after ownership checking (note that, similar to borrow checking, it does not prevent spatial memory errors such as index-out-of-range).
- To demonstrate the effectiveness of our approach, we carry out a concrete exercise for proving end-to-end safety following the pattern in Fig. 1. We implement a hash map implemented partly in Owlang and partly in C. We then verify the partial safety of the Owlang module and the total safety of the C module, respectively. By compiling the C module using CompCertO and the Owlang module using our Owlang compiler, we prove the total safety of the linked assembly code. Finally, by the soundness of open safety, we prove the regular partial correctness for final assembly code.

The formal development of this work is fully formalized in the Rocq theorem prover. In the rest of the paper, we first present the key ideas of our approach in §2. We then introduce necessary technical background §3. We present the formal framework of open safety in §4 and the Owlang compiler in §5. We discuss our application to hash map in §6. Finally, we evaluate our work and discuss related work in §7, and conclude in §8.

2 Key Ideas

We use a running example in Fig. 2 to illustrate our key ideas. It presents a snippet of the implementation of our hash map composed of a C module `buks.c` and a Owlang module `list.rs` (Owlang's syntax closely follows Rust). The two modules are compiled separately and linked together at assembly level. Our goal is to prove the safety of the target code following the pattern in Fig. 1.

The main function in Fig. 2 is `hmap_process`. Given a hash map `hmap` and a key `k`, it is responsible for encrypting the value associated with `k` in `hmap`. The encryption function `process` is provided at the C side. For simplicity of illustration, it is an XOR operation with a fixed encryption key value 42. Our hash map is implemented as an array of buckets where each bucket stores a pointer to a linked list of key-value pairs. The bucket array (of type `HashMap`) is allocated by `malloc` (not shown here) and managed at the C side, while the linked list (of type `enum List`) is implemented as an algebraic data type in Owlang and *automatically* managed by ownership semantics like in Rust. At the C side, a bucket pointing to a linked list has type `void*` (i.e., `List_ptr`), indicating that linked lists are opaque data structures to C.

Looking more closely at the implementation, we notice that `hmap_process` first obtains the pointer to the bucket of `k` by calling `find_bucket`. It then invokes the external function `find_process` in Owlang which recursively traverses the linked list until it either finds the corresponding value (then calls back to `process` at C side) or returns without finding the value. Notice that, in `list.rs`, the `match` command at line 13 takes over the ownership of the linked list stored in `l`. After processing the linked list, we need to return its ownership back to `l`, as shown at line 22. This signifies the

```c
/* buks.c */
typedef void* List_ptr; // bucket type
typedef List_ptr* HashMap; // buckets array
typedef unsigned int uint;
#define N 10 // number of buckets
// external functions implemented in Owlang
extern List_ptr find_process(List_ptr, int);
extern uint hash(int, uint);

List_ptr* find_bucket(HashMap hmap, int k){
  uint index = hash(k, N);
  return &(hmap[index]);
}

void hmap_process(HashMap hmap, int k){
  List_ptr* buk = find_bucket(hmap, k);
  if(*buk != NULL)//bucket is initialized
    *buk = find_process(*buk, k);
}

int* process(int* v){
    *v ^= 42; return v;
}
```

(a) Array of Buckets in C

```rust
/* list.rs (the same suffix as Rust) */
struct Node { key: i32, val: Box<i32>,
    next: Box<List> }
enum List { Nil, Cons(Node) }
// external function implemented in C
extern fn process(v: Box<i32>) -> Box<i32>

fn hash(k: i32, range: u32) -> u32 {
    return k % range; }

fn find_process(l: Box<List>, k: i32)
-> Box<List> {
 match *l {
  List::Nil => return l, //value not found
  List::Cons(node) => {
    if k == node.key { // value found
       node.val = process(node.val);
    } else { // find recursively
        node.next = find_process(
            node.next, k);
    } // return the ownership
    *l = List::Cons(node); return l;
} } }
```

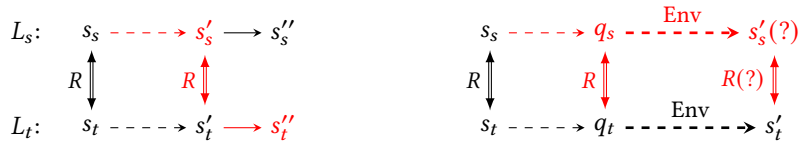
(b) Linked List in Owlang

Fig. 2. An Example of Hash Map Implemented in Owlang and C

main difference between Owlang and Rust, i.e., it only implements Rust's ownership semantics, not its reference or borrowing semantics. Hence, borrowing needs to be implemented as explicit ownership transfers which is a bit cumbersome. Despite that, Owlang's ownership semantics and compiler work very much like Rust, therefore sufficient for illustrating our approach in Fig. 1.

Our running example illustrates a typical scenario in programs combining safe and unsafe code, i.e., responsibility of managing resources is divided between safe and unsafe languages. Data structures that can be naturally described in safe languages are managed by safe code and automatically checked by verifying compilation, while the remaining data structures (e.g., array) is written in the unsafe language encapsulated under safe interfaces. For example, the linked list is implemented in the Owlang with the benefit that all memory operations (e.g., dereference of `l` at line 13 and 22 in Fig. 2b) are ensured safe by ownership checking of the Owlang compiler. Array is implemented in C as it is difficult to check the safety of array accesses statically (e.g., line 12 in Fig. 2a). The safe and unsafe modules may interact with each other in complicated ways. In our example, the two modules *mutually* invoke each other, creating a cyclic dependency between them, which significantly complicates verification of their safety.

Finally, note that the Owlang compiler can only guarantee partial safety. That is, it can detect all temporal memory errors, but no more. For example, `hash` requires that its second argument `range` is not zero to prevent division by zero at line 9 in Fig. 2b, and it must ensure that the return value is less than `range` so that array access at `buks.c` is safe. Those safety properties must be captured or checked using other methods, such as pre- and post-conditions in program logics.



(a) Preservation for Complete Programs (b) Problem with Open Modules

Fig. 3. Preservation of Partial Correctness via Backward Simulation

All the above complexity needs to be dealt with to realize the approach in Fig. 1. In the following subsections, we successively discuss the main challenges and the corresponding key ideas to solving them following the same structure of Fig. 1.1. We then conclude with an overview of our approach.

2.1 Challenge: Preserving Modular Safety by Verified Compilation

First and foremost, we need to define safety for each of the modules in Fig. 2 and utilize verified compilers to preserve them to target code. As we have discussed in the introduction, VST has successfully utilized CompCert to preserved partial correctness for *complete* programs. However, the same approach does not work for open modules. Below we analyze the fundamental problem behind this phenomenon.

The state-of-the-art verified compiler, i.e., CompCert, gives small-step operational semantics to its languages. Such a semantics is formalized as an LTS describing a transition relation $s \rightarrow s'$, meaning that execution in one-step changes the program state from s to s', and a set of initial states I. We write $s \dashrightarrow s'$ to denote a state transition in multiple steps. The correctness of a compiler pass is formalized as *backward simulation* between LTS. Given the source LTS L_s and target LTS L_t, the backward simulation $L_t \leqslant L_s$ maintains an invariant R (binary relation) between source and target states such that the initial states of L_t and L_s are related by R, and the following properties hold:

- **Small-step Simulation:** For any source state s_s and target state s_t, if $(s_s, s_t) \in R$ and $s_t \rightarrow s'_t$ for some s'_t, then $s_s \dashrightarrow s'_s$ and $(s'_s, s'_t) \in R$ for some s'_s.
- **Forward progress:** For any $(s_s, s_t) \in R$, if there is some s'_s s.t. $s_s \rightarrow s'_s$, then there exists some s'_t s.t. $s_t \rightarrow s'_t$.

The first property denotes that any one-step transition in the target is simulated by zero or more steps in the source. The second property denotes that if the source does not get stuck at some state s_s, then the target will also not in the related state s_t, i.e., at any point of execution a safe source state is related to a safe target state. A critical property of backward simulation is its transitivity:

- **Transitivity of Backward Simulation:** $\forall\ L_1\ L_2\ L_3, L_1 \leqslant L_2 \Rightarrow L_2 \leqslant L_3 \Rightarrow L_1 \leqslant L_3$.

With this property, correctness of multiple compiler passes can be combined to form the correctness of the whole compiler.

VST is a separation logic for C programs. It operates on `Clight`, a deterministic subset of C defined in CompCert. A Hoare triple $\{P\}M\{Q\}$ in VST has standard interpretation in separation logics. For simplicity of the discussion, let us focus on a complete program M, i.e., program with a `main` function whose pre-condition is trivially true (i.e., $P = \top$). We further assume that M does not terminate. In this case, $\{\top\}M\{Q\}$ is equivalent to that M is *reachable safe*, i.e., any state reachable from any initial state in I cannot get stuck. This is formally defined on the semantics L of M:

- `reach_safe` $L\ :=\ \forall\ s \in I,\ s', s \dashrightarrow s' \Rightarrow \exists\ s'', s' \rightarrow s''$.

Partial Correctness is preserved by backward simulation as follows:

- **Preservation of Safety:** $\forall\ L_t\ L_s, L_t \leqslant L_s \Rightarrow$ `reach_safe` $L_s \Rightarrow$ `reach_safe` L_t.

The key steps for proving this property are illustrated in Fig. 3a where conclusions (marked in red) are derived from assumptions (marked in black). To prove `reach_safe` L_s, assuming $s_t \dashrightarrow s_t'$, we need to prove $s_t' \rightarrow s_t''$ for some s_t''. By small-step simulation, we know $s_s \rightarrow s_s'$ and $(s_s', s_t') \in R$ for some s_s'. By `reach_safe` L_s, we know $s_s' \rightarrow s_s''$ for some s_s''. Finally, by forward progress, we conclude $s_t' \rightarrow s_t''$ for some s_t''. Therefore, in VST, partial correctness for complete programs can be preserved by the correctness of CompCert.

However, the above proof breaks for open modules. To see that, first notice that extensions to CompCert supporting open modules (i.e., VCC) [13, 21, 24, 45, 47, 53, 55] also use small-step simulations which we call *open simulations*. A key property of open simulations is that they make no assumption about simulations in the environment. In particular, if a source module and its compiled target call into the environment, there is no guarantee about the execution of environment whatsoever (e.g., if they will eventually return). Instead, simulations for open modules can resume after calling into environment *only if we can provide some source and target states after the call returns and reestablish the invariant R*. By definition of small-step simulations, this assumption can indeed be satisfied when the simulation for environment is explicitly provided.

The above change makes small-step simulation for open modules not compatible with preservation of `reach_safe`. If we try to replay the previous proof, i.e., by assuming $L_t \preccurlyeq L_s$ and `reach_safe` L_s to prove `reach_safe` L_t for open modules L_t and L_s, we get into problems as depicted in Fig. 3b. Here, we assume that the target execution starts from s_t, makes a call q_t to the environment, and reaches s_t' after the call returns. As an example, `hmap_process` may call into the external function `find_process` in Fig. 1. The goal is to show that s_t' cannot get stuck like in Fig. 3a. For this, we need to show existence of some s_s' such that $(s_s', s_t') \in R$. However, this is impossible: although we can prove the source execution reaches some q_s matching q_t by simulating internal steps, we have no idea if the call q_s into environment will ever return, since $L_t \preccurlyeq L_s$ does not make any assumption about environment as discussed in the last paragraph.

In conclusion, preservation of partial correctness fails for open modules because open simulations are small-step, they make no assumption about how the big-step execution of environment is simulated, instead relying on assumptions that such execution may eventually return with reestablished internal invariants. This is inherently in conflict with the big-step nature of partial correctness. One solution is to build the assumption about simulation in environment execution into open simulations. This is a known approach which effectively changes open simulations into *logical relations* [8, 20]. It is well-known logical relations are not transitive in general [2], indicating this change would make it difficult to support multi-pass compilers like CompCert. Because we would like to not break verified compilers, we have to find a suitable alternative to partial correctness.

2.2 Key Idea: Open Safety

The above challenge comes from the misalignment between "big-step" partial correctness and "small-step" simulations. We solve this challenge by presenting a "small-step" variant of partial correctness such that is composable at the boundary of modules and can be modularly preserved by the state-of-the-art verified compilers. We call it *open safety*. The key idea underlying *open safety* is to maintain internally an invariant for program states at any step of execution. Instead of stating safety as "any reachable state does not get stuck", we define safety as "any state satisfying the invariant does not get stuck". This change makes safety no longer dependent on big-step execution history, but only on the current state. As such, open safety aligns perfectly with the simulation property of small-step simulations, thereby enables preservation of safety under them. Below we discuss the design of open safety more concretely by building it into the framework of CompCertO [24, 55], and we show how it can be preserved by CompCertO to target code.

In CompCertO, semantics of modules are defined as *open labeled transition systems* (open LTS) which communicate with the environment through *language interfaces*. We write $L : A \twoheadrightarrow B$ to denote that the LTS L has incoming interface B (accepting function calls from environment) and outgoing interface A (initiating calls into environment). For example, $[\![\texttt{buks.c}]\!] : C \twoheadrightarrow C$ denotes that $[\![\texttt{buks.c}]\!]$ communicates with the environment using C-style function calls and returns. Compiler correctness theorems for open modules are defined as *open simulations* between open LTS with *rely-guarantee* conditions. In this paper, we make use of *open backward simulation* $L_t \leqslant_{\mathbb{R}\twoheadrightarrow\mathbb{S}} L_s$ between the source LTS L_s and target one L_t where $\mathbb{R}$ and $\mathbb{S}$ are *simulation conventions* describing the rely-guarantee conditions for outgoing and incoming calls, respectively. For simplicity of discussion, we assume that all the incoming and outgoing calls have the same language interface and use a single simulation convention $\mathbb{R}$. We write backward simulation as $L_t \leqslant_{\mathbb{R}} L_s$ in this case.

We define assertions on language interfaces, referred to as *safety interfaces*, to capture the pre- and post-conditions of partial correctness. Later, we show that simulation conventions and encapsulation for composing safe and unsafe code can also be described as safety interfaces. Throughout the paper, we denote safety interfaces using blackboard bold letters (e.g., $\mathbb{P}$, $\mathbb{Q}$ and $\mathbb{I}$). We write the following notation to represent *open safety* of an LTS L:

$$\textbf{Open Safety:} \quad L \models \mathbb{P} \twoheadrightarrow \mathbb{Q}$$

Here, $\mathbb{P}$ specifies the pre- and post-conditions for the external functions of L and $\mathbb{Q}$ specifies those for the internal functions of L which may be invoked by the environment. Open safety is *composable* under semantic linking of two LTS, as long as they use complementary safety interfaces (see §4.2).

Roughly speaking, a safety interface $\mathbb{P}$ denotes a pre-condition P' and a post-condition P''. Intuitively, $L \models \mathbb{P} \twoheadrightarrow \mathbb{Q}$ may be interpreted as the Hoare triple $\{Q'\}L\{Q''\}$ which depends on an assumption $\{P'\}L_e\{P''\}$ where L_e represents the environment of L. However, this intuition is not entirely accurate as $L \models \mathbb{P} \twoheadrightarrow \mathbb{Q}$ is stronger than Hoare triples in the following aspects. First, the LTS L is an abstract notation agnostic of any particular language while Hoare triples are usually defined against concrete languages. Second, the small-step nature of L makes it agnostic of its environment, which may even calls back to L. Therefore, it is very easy to encode mutually recursive safety requirements in $L \models \mathbb{P} \twoheadrightarrow \mathbb{Q}$. Finally and most importantly, an invisible (existentially quantified) invariant *Inv* is maintained during the execution of L s.t. the following properties hold:

- **Invariant under Transition:** $\forall\ s\ s', s \in \mathit{Inv} \Rightarrow s \rightarrow s' \Rightarrow s' \in \mathit{Inv}$;
- **Progress under Invariant:** $\forall\ s, s \in \mathit{Inv} \Rightarrow s \in \mathit{progress}$

The first property make sure *Inv* holds throughout execution. The second property make sure that, in any state satisfying *Inv*, execution either ends safely or can still make progress (denoted by $s \in \mathit{progress}$). Together they enforce the safety of L in a small-step fashion.

It is worth noting that the above properties about *Inv* exactly mirrors type preservation and progress properties in type systems. Therefore, this invariant can be viewed as the "type" over states. On the other hand, unlike types, our invariants are not visible outside of a module as they are *existentially quantified* (that is why it does not appear in the notation $L \models \mathbb{P} \twoheadrightarrow \mathbb{Q}$). This has two benefits. First, invariants can be established without relying on any concrete type systems. Second, open safety for modules with different internal invariants remain composable. Therefore, the introduction of invariant is quite lightweight and non-intrusive.

The small-step nature of open safety makes it preservable under open simulations:

$$\textbf{Safety Preservation:} \quad L_s \models \mathbb{P} \twoheadrightarrow \mathbb{Q} \Rightarrow L_t \leqslant_{\mathbb{R}} L_s \Rightarrow L_t \models \mathbb{P} \triangleleft \mathbb{R} \twoheadrightarrow \mathbb{Q} \triangleleft \mathbb{R}$$

The simulation convention (i.e., $\mathbb{R}$) is encoded into the target safety interfaces via the $\triangleleft$ operator (§4.3). Intuitively, $\mathbb{R}$ captures the calling convention of the compiler, which helps translating safety

interface for the source module into the target. For example, the target safety may depend on that callee-saved registers are unchanged by external calls, which is captured in $\mathbb{R}$.

Finally, we would like to point out that open safety is closely related to existing safety definitions: the only significant addition is internal invariants which already takes the form of types in type systems and is almost always constructed (but hidden in the final result) when proving partial correctness. Therefore, although we do not connect with any source-level verification tools in this work, we conjecture this connection would be quite nature.

2.3 Challenge: Mixing Safety Checking with Safety Preservation

Verifying compilation (e.g., borrow/ownership checking) checks a certain type of errors (e.g., temporal memory errors), making a checked program *partially safe*. To preserve this partial safety by semantics-preserving compilers, a naive approach is to prove that if the source is absent of this type of errors, then so is the target. However, this approach does not work in an optimizing compiler. Consider the following snippet of C code:

```
1   int x = 10; int *p;    x/0; *p = 10;
```

Although `p` is an uninitialized pointer, this code does not contain any memory error as the execution crashes at `x/0` before dereferencing `p`. Since division by zero is an undefined behavior, an optimizing compiler may compile `x/0` into arbitrary code, say a `Nop`. As a result, the optimized code now executes `*p = 10` which causes a memory error.

This example shows that even if a source program is absent of a certain class of errors, compiler optimizations may reintroduce them. As discussed in §1.1.2, existing approaches to formalizing verifying compilation does not deal with compilation or optimization on partially checked programs. Therefore, we need a way to formalize partial safety that can cooperate with verified compilation.

2.4 Key Idea: Open Safety Parameterized by Errors

To solve the above challenge, we generalize open safety to account for safety guarantees that *exclude* certain safety properties. We refer to it as *open partial safety* or just partial safety.

The key idea is to relax the "make progress" requirement in open safety. Specifically, if a module is partially safe, its execution can either makes progress in a normal way or enters states that violate given safety properties, e.g., states whose next steps may cause memory errors by accessing freed memory. Formally, we use $L \models^{E} \mathbb{P} \twoheadrightarrow \mathbb{Q}$ to denote partial safety, where E is a state predicate that specifies error states, i.e., states that violate certain safety properties. For example, we write $L \models^{\texttt{mem-err}} \mathbb{P} \twoheadrightarrow \mathbb{Q}$ to represent that L may end execution in states that are *not* memory safe. The old total safety is thus defined as $L \models^{\emptyset} \mathbb{P} \twoheadrightarrow \mathbb{Q}$; we often omit $\emptyset$ for simplicity. The incremental verification is achieved by first verifying the partial safety $[\![M]\!] \models^{E} \mathbb{P} \twoheadrightarrow \mathbb{Q}$ for the source modules M, and then using verifying compilation to ensure that M cannot reach any state in E. This establishes the correctness theorem for verifying compilation as follows.

Correctness of Verifying Compilation: $[\![M]\!] \models^{E} \mathbb{P} \twoheadrightarrow \mathbb{Q} \Rightarrow \texttt{Check}(M) \Downarrow \Rightarrow [\![M]\!] \models \mathbb{P} \cdot \mathbb{I} \twoheadrightarrow \mathbb{Q} \cdot \mathbb{I}$

Here, $(\_ \cdot \_)$ is the conjunction of safety interfaces. We write $\texttt{Check}(M) \Downarrow$ to indicate that M passes the verifying pass. After verifying compilation, the error states predicate E is ruled out from the checked module. Moreover, the checked module automatically satisfies a safety interface $\mathbb{I}$, which serves as the safe encapsulation that specifies how this module can safely interact with other modules, e.g., how modules written in safe Rust interact with unsafe modules. Note that the above conclusion can be generalized to partial safety. For instance, we may establish $[\![M]\!] \models^{E_1} \mathbb{P} \cdot \mathbb{I} \twoheadrightarrow \mathbb{Q} \cdot \mathbb{I}$ where $E_1 \subset E$, deferring the responsibility of ruling out E_1 to subsequent verifying passes.

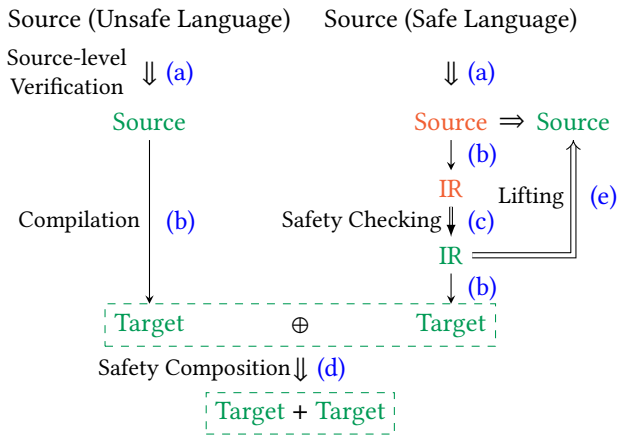


Fig. 4. The Framework (Green: total safety, Red: partial safety)

2.5 An Overview of Our Framework

The complete framework is depicted in Fig. 4 with blue labels for each step of verification. From the top to bottom, we have two source modules written in unsafe and safe languages, respectively. Their safety is verified at the source level (a). For the module in unsafe language, it is verified to be totally safe (colored in green). The total safety requires that the execution of module never gets stuck, i.e., has no undefined behaviors. For the module in safe language, it is verified to be partially safe (colored in red). The partial safety is defined by *allowing* the modules to step to the states that violate certain safety properties (e.g., it may step to memory error states). These violation will be proved impossible by the safety checking passes. The two source modules are compiled to IRs and then to the same target (e.g., assembly) for linking. The compilation (b) in our framework preserves the total and partial safety from Source to IR and then to Target. The safety checking (c) performed on the IR of the module in safe language ensures that the partially safe module is totally safe. Moreover, the total safety verified in IR can be propagated to the source module (e), which is useful for further refinement proofs at the source-level. Finally, the safety composition (d) ensures that the syntactically linked target module is totally safe. If this target is a complete program, we can derive regular partial correctness from total safety.

An an example, the proof of our running example in Fig. 2 proceeds as follows. We first prove that `buks.c` is totally safe and `list.rs` is partially safe. We then prove the total safety of `list.ir` by ownership checking. We then prove the total safety of the target assembly modules `list.s` and `buks.s` by semantics preservation of CompCertO and the Owlang compiler. We apply horizontal compositionality and the safety preservation under the syntactic linking to prove `list.s + buks.s` is totally safe, from which we finally derive the partial correctness of `list.s + buks.s`. We shall discuss the details in §6.

3 Background

3.1 Closed Semantics and Reachability Safety

In CompCert [29, 30], the semantics of a whole program, which we refer to as *closed semantics*, can be represented by $L = \langle S, I, \rightarrow, F \rangle$, consisting of a set of states S, initial states $I \subseteq S$, an internal transition relation $\rightarrow \subseteq S \times \mathcal{E} \times S$ and final states $F \subseteq S \times \texttt{int}$ that return integers. We often omit the *event* ($\mathcal{E}$) of the internal transition in closed semantics and the following modular semantics.

In closed semantics, *reachability safety* is a predicate defined on LTS as follows:

Definition 3.1 (Reachability Safety). An LTS L is reachable safe, denoted as $\texttt{reach\_safe}\ L$, if:

$$\forall\ s\ s',\ s \in I \Rightarrow s \dashrightarrow s' \Rightarrow (\exists\ s'',\ s' \rightarrow s'') \vee (\exists\ r,\ (s', r) \in F)$$

Here, $\dashrightarrow$ denotes the multi-steps transition. In this paper, we also write $\texttt{reach\_safe}(s)$ to denote that the state s is reachable safe, i.e., all reachable states from s either can take a next step or it is in the final state. In the above definition, we do not specify the postcondition for the return value r, which is different from the definition of partial correctness as mentioned in §1.1. We will see in §4.4 that proving a whole program is reachable safe also requires the proofs of partial correctness for this program, so in this paper reachability safety is the final goal of our verification.

3.2 Semantics for Open Modules

We use *open labeled transition systems* (open LTS) from CompCertO [24] to define semantics of modules. The open LTS extends CompCert's LTS by characterizing modules' interaction with environments through explicit transition relations that are parameterized by *language interfaces*.

A language interface $A = \langle A^q, A^r \rangle$ consists of a set of queries A^q and replies A^r. The language interface used in the C module is $C = \langle \texttt{val} \times \texttt{sig}_C \times \texttt{val}^* \times \texttt{mem}, \texttt{val} \times \texttt{mem} \rangle$. Here $\texttt{val}$, $\texttt{mem}$ and $\texttt{sig}_C$ are the types of values, memory and the function signature of the callee. The queries of C have the form $v_f[sg](\vec{v})@m$ where v_f is the function pointer points to the callee, sg is the signature, $\vec{v}$ is the list of arguments and m is the memory. The replies have the form $v'@m'$ containing the return value and the memory. The assembly interface is $\mathcal{A} = \langle \texttt{regset} \times \texttt{mem}, \texttt{regset} \times \texttt{mem} \rangle$ where its queries and replies take the form $rs@m$ containing the registers set and memory.

The open LTS, denoted by $L : A \twoheadrightarrow B$, is a tuple $\langle S, I, \rightarrow, F, X, Y \rangle$ where A (B) is the language interface for outgoing (incoming) calls and returns. The transition relation $I \subseteq B^q \times S$ receives an incoming query to initialize the LTS; $\rightarrow \subseteq S \times \mathcal{E} \times S$ is the internal transition relation where $\mathcal{E}$ is the event type that is often omitted in our discussion; $F \subseteq S \times B^r$ finalizes the LTS with an outgoing reply; $X \subseteq S \times A^q$ models outgoing queries (i.e., external calls) from the external states; $Y \subseteq S \times A^r \times S$ defines how an external state continues execution by selecting a resumed state based on incoming replies (i.e., the return values of external calls). To facilitate the following discussion, we first formalize the *progress property* for an internal state:

Definition 3.2 (Progress Property). Given $L : A \twoheadrightarrow B$ and a state $s \in S$, $s \in \mathit{progress}$ holds if:

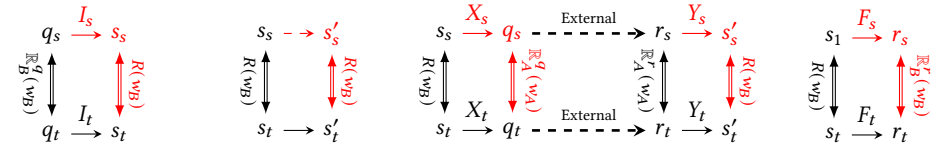
$$(\exists\ s',\ s \rightarrow s') \vee (\exists\ r,\ (s, r) \in F) \vee (\exists\ q,\ (s, q) \in X)$$

The first and the second clauses in the progress property are the same as the conclusion in Definition 3.1, while the third clause ensures that external states can make progress by emitting some query. We can then define reachable safe states as $\texttt{reach\_safe}(s) = \forall\ s', s \dashrightarrow s' \Rightarrow s' \in \mathit{progress}$.

Semantics Linking. Two open LTS with compatible interfaces, i.e., $L_1, L_2 : A \twoheadrightarrow A$, can be composed as $L_1 \oplus L_2 : A \twoheadrightarrow A$. In the composed LTS, the internal state $s = s_{i_1} :: \ldots :: s_{i_n}$ is a stack of states from L_1 and L_2 (i.e., $s_{i_k} \in S_1$ or $s_{i_k} \in S_2$) to simulate the mutual invocation between L_1 and L_2. The key part of the semantic linking is to merge X (Y) of one LTS to I (F) of another into an internal step of the composed semantics. Therefore, safety composition must ensure that the conditions required by one module for making external calls (i.e., X) satisfy the assumptions expected by the other module's initial transition (i.e., I), and similarly for returns from one's F to Y of another.

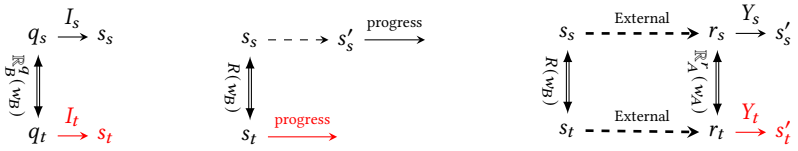
3.3 Open Backward Simulations

In CompCertO [24], simulations between open LTS is defined as *open simulations*, where the simulation relation is extended to a *Kripke relation* used to describe the evolution of program states.



(a) Initial states (b) Internal steps (c) External calls (d) Final states

Fig. 5. Simulation Diagrams for Open Backward Simulation



(a) Initial progress (b) Internal progress (c) External progress

Fig. 6. Forward Progress Property of Open Backward Simulation

A Kripke relation $R : W \to \{S \mid S \subseteq A \times B\}$ is a family of relations indexed by a *Kripke world* W and we define $\mathcal{K}_W(A, B) = W \to \{S \mid S \subseteq A \times B\}$. The evolution of Kripke worlds is described by the accessibility relation, which is denoted as $w \rightsquigarrow w'$ for some w and w'. At the boundary of open LTS, the simulation is expressed by *simulation conventions* which relate source and target language interfaces A_1 and A_2. The simulation convention is defined as a tuple $\mathbb{R} = \langle W, \mathbb{R}^q : \mathcal{K}_W(A_1^q, A_2^q), \mathbb{R}^r : \mathcal{K}_W(A_1^r, A_2^r)\rangle$ which we write as $\mathbb{R} : A_1 \Leftrightarrow A_2$. For some optimization passes, CompCertO uses unary simulation conventions to prove their correctness by establishing invariants on the interfaces. As we will see, these unary simulation conventions are well-suited for our safety definition.

The compilation correctness of modules is defined as *open backward simulation* denoted as $L_t \leqslant_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_s$ (or $L_t \leqslant_{\mathbb{R}_A} L_s$ if $\mathbb{R}_A = \mathbb{R}_B$). It is formally defined as follows [1]:

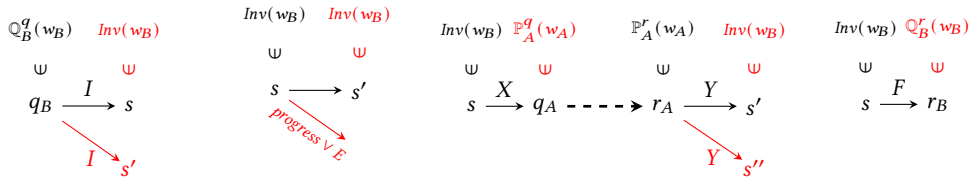
Definition 3.3 (Open Backward Simulation). Given $L_s : A_s \twoheadrightarrow B_s$, $L_t : A_t \twoheadrightarrow B_t$, $\mathbb{R}_A : A_s \Leftrightarrow A_t$ and $\mathbb{R}_B : B_s \Leftrightarrow B_t$, $L_t \leqslant_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_s$ holds if there is some Kripke relation $R \in \mathcal{K}_{W_B}(S_s, S_t)$ that satisfies:

(1) $\forall\ w_B\ q_s\ q_t\ s_t,\ (q_s, q_t) \in \mathbb{R}_B^q(w_B) \Rightarrow (q_t, s_t) \in I_t \Rightarrow \exists\ s_s, (s_s, s_t) \in R(w_B) \wedge (q_s, s_s) \in I_s.$
(2) $\forall\ w_B\ s_s\ s_t,\ (s_s, s_t) \in R(w_B) \Rightarrow s_t \to s_t' \Rightarrow \exists\ s_s',\ s_s \dashrightarrow s_s' \wedge (s_s', s_t') \in R(w_B).$
(3) $\forall\ w_B\ s_s\ s_t\ q_t,\ (s_s, s_t) \in R(w_B) \Rightarrow (s_t, q_t) \in X_t \Rightarrow$
$\exists w_A\ q_s,\ (q_s, q_t) \in \mathbb{R}_A^q(w_A) \wedge (s_s, q_s) \in X_s \wedge$
$\forall\ r_s\ r_t\ s_t', (r_s, r_t) \in \mathbb{R}_A^r(w_A) \Rightarrow (s_t, r_t, s_t') \in Y_t \Rightarrow \exists\ s_s', (s_s, r_s, s_s') \in Y_s \wedge (s_s', s_t') \in R(w_B).$
(4) $\forall\ w_B\ s_s\ s_t\ r_t,\ (s_s, s_t) \in R(w_B) \Rightarrow (s_t, r_t) \in F_t \Rightarrow \exists\ r_s, (s_s, r_s) \in F_s \wedge (r_s, r_t) \in \mathbb{R}_B^r(w_B).$
(5) The below forward progress property holds.

Definition 3.4 (Forward Progress Property). Given $L_s : A_s \twoheadrightarrow B_s$, $L_t : A_t \twoheadrightarrow B_t$, $\mathbb{R}_A : A_s \Leftrightarrow A_t$, $\mathbb{R}_B : B_s \Leftrightarrow B_t$, and the simulation relation R, the forward progress property holds if:

(1) $\forall\ w_B\ q_s\ q_t\ s_s,\ (q_s, q_t) \in \mathbb{R}_B^q(w_B) \Rightarrow (q_s, s_s) \in I_s \Rightarrow \exists\ s_t,\ (q_t, s_t) \in I_t.$
(2) $\forall\ w_B\ s_s\ s_t,\ (s_s, s_t) \in R(w_B) \Rightarrow \texttt{reach\_safe}(s_s) \Rightarrow s_t \in \mathit{progress}$
(3) $\forall\ w_B\ w_A\ r_s\ r_t\ s_s\ s_s'\ s_t, (s_s, s_t) \in R(w_B) \Rightarrow (r_s, r_t) \in \mathbb{R}_A^r(w_A) \Rightarrow (s_s, r_s, s_s') \in Y_s \Rightarrow$
$\exists\ s_t', (s_t, r_t, s_t') \in Y_t.$

[1] For simplicity, we omit the requirements of $\texttt{reach\_safe}(s_s)$ in (2), (3) and (4), as these requirements can be simply derived in the proof of safety preservation by the safety of L_s.



(a) Initial States (b) Internal Steps (c) External Calls and Returns (d) Final States

Fig. 7. Diagrams for the Properties of Open Safety

Properties (1) to (4) describe how the source LTS simulates the target one, each of whose simulation diagram is depicted in Fig. 5 (in the same order). Property (5) is the forward progress property introduced in §2.1 but is extended for the open semantics. The three clauses of forward progress properties are depicted in Fig. 6.

In Fig. 5 and Fig. 6, black arrows and symbols represent assumptions and red ones represent conclusions. In the open backward simulation, property (1) requires $R(w_B)$ holds for the initial states initialized by matched queries; (2) requires $R(w_B)$ to be preserved during the execution; (3) requires that related external states emit related queries to the environment, and if the environment return related replies, then $R(w_B)$ can be reestablished; (4) requires that the final replies are related; (5) requires that when the source and target are matched–either by matching queries/replies or by matching internal states–the safety of the source LTS can be transferred to the target.

4 End-to-end Safety for Verified and Verifying Compilation

4.1 Definition of Open Safety

We first introduce *safety interfaces*, which are the unary simulation conventions in CompCertO. These conventions, originally used to prove compiler correctness, are well-suited for expressing the boundary assertions required by our open safety definition. Specifically, given a language interface A, the safety interface for A is a tuple $\mathbb{I}_A = \langle W, \mathbb{I}_A^q : \mathcal{K}_W(A^q), \mathbb{I}_A^r : \mathcal{K}_W(A^r) \rangle$ which we write as $\mathbb{I}$ if A can be inferred from the context. In our safety definition, the component $\mathbb{I}_A^q$ specifies the preconditions of incoming or outgoing calls, while $\mathbb{I}_A^r$ specifies the postconditions of returns to or from the environment. The Kripke relation $\mathcal{K}_W(A) = W \to \{S \mid S \subseteq A\}$ is essential to capture the rely-guarantee conditions in verified and verifying compilation. The rely-guarantee conditions are similar to propositions in post conditions of Hoare triples in separation logic [50] which describes changes to states w.r.t. pre-conditions. For example, if a module L emits an external call satisfying $\mathbb{I}_A^q(w_A)$ for some Kripke world w_A, it can assume that the return (e.g., the return memory) from the external call must satisfy $\mathbb{I}_A^r(w_A)$. Note that the accessibility relation is often implicit in $\mathbb{I}_A^r$, i.e., $r \in \mathbb{I}_A^r(w) = \exists\, w', w \rightsquigarrow w' \wedge r \in \mathbb{I}_A^r(w')$. The accessibility relation $w \rightsquigarrow w'$ serves to enforce invariants across possible worlds, e.g., to ensure that some memory owned by L is protected, which corresponds to the frame-preserving update in separation logics We will revisit this usage in our definition of the safety interface required by the ownership checking in §5.1.

We define open safety that is parametric in a state predicate E. By instantiating the predicate to specific error states, we can obtain total safety (by setting E to $\emptyset$) and partial safety. The assertions at the boundaries are specified by the safety interfaces.

Definition 4.1 (Open Safety). Given $L : A \twoheadrightarrow B, \mathbb{P}_A, \mathbb{Q}_B$ and a state predicate $E \subseteq S$, $L \models^E \mathbb{P}_A \twoheadrightarrow \mathbb{Q}_B$ holds if there is some invariant $Inv \in \mathcal{K}_{W_B}(S)$ that satisfies:

(1) $\forall\ w_B\ q_B, q_B \in \mathbb{Q}_B^q(w_B) \Rightarrow (\exists\ s,\ (q_B, s) \in I) \wedge (\forall\ s,\ (q_B, s) \in I \Rightarrow s \in Inv(w_B))$.

(2) $\forall\ w_B\ s\ s',\ s \in Inv(w_B) \Rightarrow s \to s' \Rightarrow s' \in Inv(w_B)$.

(3) $\forall\ w_B\ s\ q_A,\ s \in Inv(w_B) \Rightarrow (s, q_A) \in X \Rightarrow \exists\ w_A,\ q_A \in \mathbb{P}_A^q(w_A) \wedge$
$\forall\ r_A,\ r_A \in \mathbb{P}_A^r(w_A) \Rightarrow (\exists\ s',\ (s, r_A, s') \in Y) \wedge (\forall\ s',\ (s, r_A, s') \in Y \Rightarrow s' \in Inv(w_B))$.
(4) $\forall\ w_B\ s\ r_B,\ s \in Inv(w_B) \Rightarrow (s, r_B) \in F \Rightarrow r_B \in \mathbb{Q}_B^r(w_B)$.
(5) $\forall\ w_B\ s,\ s \in Inv(w_B) \Rightarrow s \in progress \vee s \in E$.

Open safety requires the LTS to be equipped with an invariant, which we refer to as *safety invariant*. Depending on the verification techniques, the invariant may take different forms—for instance, type information, Hoare triples, or state invariant established via model checking. The properties of open safety are depicted in Fig. 7, where black symbols represent assumptions and red ones represent conclusions. For each properties, the open safety requires that: (1) L can enter an initial state if the query satisfies the precondition $\mathbb{Q}_B^q(w_B)$ and every initial state from this query satisfies the invariant Inv; (2) Inv is preserved by each internal step; (3) if the state satisfying Inv can emit an external query, then this query satisfies the precondition $\mathbb{P}_A^q(w_A)$, and if the environment returns a reply satisfying the postcondition $\mathbb{P}_A^r(w_A)$, then L continues with this reply and all continued states from this reply satisfy Inv; (4) the final reply must satisfy the postcondition $\mathbb{Q}_B^r(w_B)$; (5) every state satisfying Inv can make progress or is in the error states specified by E.

The "small-step" nature of open safety—maintaining invariant at each small-step, makes it structurally aligned with the open simulation (i.e., Definition 3.3), where the invariant plays a role analogous to the simulation relation and both must be preserved throughout the module's execution. The open safety also captures the progress property, both at the module boundaries by safety interfaces, and during its internal execution by $s \in progress$, corresponding to the forward progress in backward simulation. The cooperation between invariant preservation and progress, just like the type preservation and progress, ensure that open safety implies partial correctness.

4.1.1 Partial Safety and Correctness of Verifying Compilation. The partial safety is an instance of open safety such that $E \neq \emptyset \wedge E \cap progress = \emptyset$. Theoretically, verifying partial safety is easier than verifying total safety, since when the program enters a state in E, the make progress requirement is trivially satisfied. The error states predicate E in partial safety can be refined by safety checking when it guarantees that a subset of error states $E_1 \subset E$ is unreachable. This refinement is considered as the correctness criterion for this safety checking pass, as shown below:

Definition 4.2 (Correctness of Verifying Compilation). Given a module M where $[\![M]\!] : A \twoheadrightarrow B$, $\mathbb{P}_A$, $\mathbb{Q}_B$, E, and safety interfaces $\mathbb{I}_A$ and $\mathbb{I}_B$ for the verifying compilation, the safety checking Check which rules out $E_1 \subseteq E$ is correct if:

$$[\![M]\!] \models^{E} \mathbb{P}_A \twoheadrightarrow \mathbb{Q}_B \Rightarrow \mathsf{Check}(M) \Downarrow \Rightarrow [\![M]\!] \models^{E \setminus E_1} \mathbb{P}_A \cdot \mathbb{I}_A \twoheadrightarrow \mathbb{Q}_B \cdot \mathbb{I}_B.$$

Here, $\mathbb{P} \cdot \mathbb{I} = \langle W_{\mathbb{P}} \times W_{\mathbb{I}}, \mathbb{P}^q \cdot \mathbb{I}^q, \mathbb{P}^r \cdot \mathbb{I}^r \rangle$ where for any q, $q \in \mathbb{P}^q \cdot \mathbb{I}^q(w_{\mathbb{P}}, w_{\mathbb{I}}) \Leftrightarrow q \in \mathbb{P}^q(w_{\mathbb{P}}) \wedge q \in \mathbb{I}^q(w_{\mathbb{I}})$ (similarly for $\mathbb{P}^r \cdot \mathbb{I}^r$). The safety interfaces $\mathbb{I}_A$ and $\mathbb{I}_B$ are the formalized assumptions of this verifying pass, e.g., type interfaces for type checker and aliasing discipline for the Rust borrow checker.

4.2 Compositionality of Open Safety

We focus on the compositionality of total safety, as safety composition in practice typically occurs at the target level between modules that have already been individually verified. In contrast, composing partial safety would require explicitly combining the error states of the modules, which complicates the safety checking.

Theorem 4.3 (Compositionality of Open Safety). *Given $L_1, L_2 : A \twoheadrightarrow A$, $\mathbb{P}_A$ and $\mathbb{Q}_A$,*

$$L_1 \models \mathbb{P}_A \twoheadrightarrow \mathbb{Q}_A \Rightarrow L_2 \models \mathbb{Q}_A \twoheadrightarrow \mathbb{P}_A \Rightarrow L_1 \oplus L_2 \models \mathbb{P}_A \uplus \mathbb{Q}_A \twoheadrightarrow \mathbb{P}_A \uplus \mathbb{Q}_A.$$

Since there may be mutual invocations between L_1 and L_2, the outgoing safety interface of one (e.g., $\mathbb{Q}_A$ of L_2) and the incoming interface of the other (e.g., $\mathbb{Q}_A$ of L_1) are often assumed to be

identical. More generally, this requirement can be relaxed: it suffices for a stronger $\mathbb{Q}'_A$ interface of L_2 to logically entail $\mathbb{Q}_A$ of L_1 by the refinement of safety interface (see §6).

The safety interface for $L_1 \oplus L_2$ is the disjoint union $\uplus$ between the interfaces of two modules:

$$\mathbb{P} \uplus \mathbb{Q} = \langle W_{\mathbb{P}} + W_{\mathbb{Q}}, \mathbb{P}^q \uplus \mathbb{Q}^q : \mathcal{K}_{W_{\mathbb{P}}+W_{\mathbb{Q}}}(A^q), \mathbb{P}^r \uplus \mathbb{Q}^r : \mathcal{K}_{W_{\mathbb{P}}+W_{\mathbb{Q}}}(A^r) \rangle.$$

Here + is the notation for building a sum type. For any q, $q \in \mathbb{P}^q \uplus \mathbb{Q}^q(w_{\mathbb{P}\uplus\mathbb{Q}}) \Leftrightarrow (w_{\mathbb{P}\uplus\mathbb{Q}} = \texttt{inl}\ w_{\mathbb{P}} \Rightarrow q \in \mathbb{P}^q(w_{\mathbb{P}})) \wedge (w_{\mathbb{P}\uplus\mathbb{Q}} = \texttt{inr}\ w_{\mathbb{Q}} \Rightarrow q \in \mathbb{Q}^q(w_{\mathbb{Q}}))$ (similarly for $\mathbb{P}^r \uplus \mathbb{Q}^r$), where `inl` and `inr` are the two constructors for the sum type. The disjoint union is used to distinguish whether a call targets L_1 or L_2, so that the corresponding incoming interface can be applied; the same applies to outgoing calls/returns. When composing multiple modules, this disjoint union can often be simplified by using unique function names to distinguish different modules, thereby avoiding explicit tagging.

For the proof of this compositionality theorem, by the definition of open safety, the key is to construct a safety invariant for $L_1 \oplus L_2$. Similar to the disjoint union of safety interfaces at module boundaries, the internal safety invariant is chosen based on the currently executing module, i.e., Inv_1 (the invariant of $L_1 \models \mathbb{P}_A \twoheadrightarrow \mathbb{Q}_A$) when L_1 is active, and Inv_2 when L_2 is. Specifically, this internal invariant is structured as a stack of invariants selected from Inv_1 and Inv_2, reflecting the control flow between L_1 and L_2 (§3.2). As we can see, the composition does not rely on the internal definitions of Inv_1 and Inv_2. This allows results from different verification techniques to be composed at the level of open safety, as long as they conform to safety interfaces at boundaries.

4.3 Preservation of Open Safety through Open Simulation

We show that open total safety can be preserved without modifying existing compiler correctness proofs (i.e., reusing the proof of Definition 3.3), as illustrated below:

Theorem 4.4 (Preservation of Open Total Safety). *Given* $L_s : A_s \twoheadrightarrow B_s$, $L_t : A_t \twoheadrightarrow B_t$, $\mathbb{R}_A : A_s \Leftrightarrow A_t$, $\mathbb{R}_B : B_s \Leftrightarrow B_t$, $\mathbb{P}_{A_s}$ *and* $\mathbb{Q}_{B_s}$,

$$L_s \models \mathbb{P}_{A_s} \twoheadrightarrow \mathbb{Q}_{B_s} \Rightarrow L_t \leqslant_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_s \Rightarrow L_t \models \mathbb{P}_{A_s} \triangleleft \mathbb{R}_A \twoheadrightarrow \mathbb{Q}_{B_s} \triangleleft \mathbb{R}_B.$$

We first introduce how to encode the simulation conventions $\mathbb{R}$ into the target safety interfaces by the $\triangleleft$ operator. Intuitively, if we view $\mathbb{R}$ as the calling convention of the compiler, a target query q_t satisfies $\mathbb{P} \triangleleft \mathbb{R}$ if it can *simulate* a source query q_s such that they are related by this calling convention and q_s satisfies $\mathbb{P}$. Therefore, we define $\triangleleft$ as follows:

Definition 4.5 ($\triangleleft$ operator). Given source and target interfaces A_s and A_t, simulation convention $\mathbb{R} : A_s \Leftrightarrow A_t$ such that $\mathbb{R} = \langle W_{st}, \mathbb{R}^q, \mathbb{R}^r \rangle$, and source safety interface $\mathbb{P} = \langle W_s, \mathbb{P}^q, \mathbb{P}^r \rangle$:

$$\mathbb{P} \triangleleft \mathbb{R} := \langle W_s \times W_{st}, \mathbb{P}^q \triangleleft \mathbb{R}^q : \mathcal{K}_{W_s \times W_{st}}(A_t^q), \mathbb{P}^r \triangleleft \mathbb{R}^r : \mathcal{K}_{W_s \times W_{st}}(A_t^r) \rangle \quad \text{where}$$

$$q_t \in \mathbb{P}^q \triangleleft \mathbb{R}^q(w_s, w_{st}) \Leftrightarrow \exists\ q_s,\ (q_s, q_t) \in \mathbb{R}^q(w_{st}) \wedge q_s \in \mathbb{P}^q(w_s)$$

$$r_t \in \mathbb{P}^r \triangleleft \mathbb{R}^r(w_s, w_{st}) \Leftrightarrow \exists\ r_s,\ (r_s, r_t) \in \mathbb{R}^r(w_{st}) \wedge r_s \in \mathbb{P}^r(w_s).$$

Here, when we consider incoming safety interfaces (dually for outgoing interfaces), the $\exists$ quantifiers in pre-conditions (i.e., $\mathbb{P}^q \triangleleft \mathbb{R}^q$) will appear in the premise of the open safety judgement, behaving like a $\forall$. It means that the environment has picked a right (i.e., adhere to the calling convention) query satisfying $\mathbb{P}^q$. The $\exists$ quantifiers in post-conditions (i.e., $\mathbb{P}^r \triangleleft \mathbb{R}^r$) will appear in the conclusion, meaning that the modules (i.e., the provers) must choose a right reply satisfying $\mathbb{P}^r$.

We illustrate the proof structure of the open safety preservation in Fig. 8. Here, each subfigure corresponds to specific clauses in Definition 4.1, where Fig. 8b includes clause (2) and (5). In each subfigure, red symbols and arrows represent conclusions (or intermediate steps) we need to prove, and black symbols and arrows represent premises (or direct consequences of the premises).

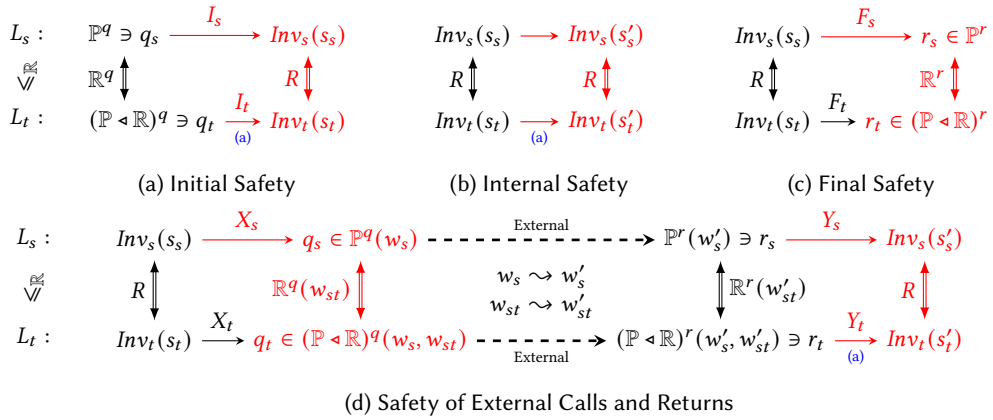


Fig. 8. Preservation of Open Safety through Open Backward Simulation

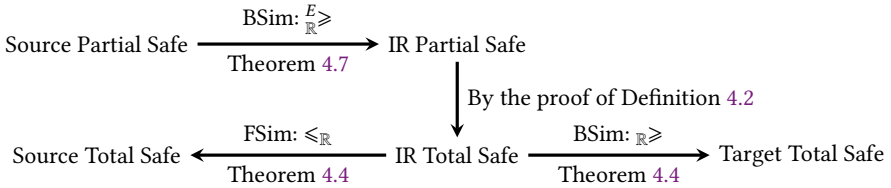


Fig. 9. A Road Map of Safety Propagation in Compilation Chain

We define the safety invariant for L_t as Inv_t such that $\forall\ s_t \in Inv_t \Leftrightarrow \exists\ s_s,\ s_s \in Inv_s \wedge (s_s, s_t) \in R$ where R is the simulation relation. Once $s_t \in Inv_t$ (or $Inv_t(s_s)$) holds, we can apply the forward progress (Fig. 6) to conclude that s_t can make progress (`reach_safe` s_s is proved by the safety of L_s), thereby proving the steps annotated by (a). To show that the invariant Inv_t is preserved during the execution, we apply the properties in Fig. 5. Now the steps annotated by (a) become assumptions. Consider the proof of Fig. 8b. Given that s_t steps to s'_t, the step simulation ensures that s_s steps to s'_s and $(s'_s, s'_t) \in R$ holds. By the preservation of Inv_s and the definition of Inv_t, we conclude $s'_t \in Inv_t$.

Although the definition of Inv_t introduces the compiler-level invariant (i.e., the simulation relation) into the internal invariant of the target module, this does not affect modular verification in practice. As discussed earlier, the composition of modules does not require knowledge of the internal structure of Inv_t. It relies solely on complementary safety interfaces between modules. Although the preserved safety interface on the target side may still carry compiler-specific calling conventions, it is entirely possible to refine it into a native safety interface (i.e., does not rely on $\triangleleft$) that encodes only the requirements imposed by the calling convention on the target program.

Based on the above proof structure, we now revisit how our open safety addresses the challenge discussed in §2.1, i.e., how to reestablish the simulation relation after external calls. First, we use $\triangleleft$ to constrain the environment so that any target reply it returns corresponds to a source reply. However, this condition alone is insufficient. If we use partial correctness to quantify the related replies, e.g., via universal quantification, they may deviate from what the open simulation expects, i.e., $\mathbb{R}^r(w'_{st})$ in Fig. 8d. We resolve this with the safety invariant that aligns with the simulation, to ensure the environment's replies are consistent with the requirements of open simulation.

4.3.1 Preservation of Open Partial Safety. As safety checking (e.g., borrow checking) is usually performed at intermediate stages, it is essential to preserve partial safety to these stages, ensuring that the safety checking can indeed verify the intended properties. The key to preserving partial safety lies in ensuring that the compiler does not miscompile error states specified by E in the source-level partial safety, into different kinds of target-level errors. For instance, a memory error should not be transformed into a division-by-zero error. Such transformations are not ruled out by backward simulation. Note that this problem (i.e., preserving certain errors) is different from that discussed in §2.3, where we discuss the difficulties of preserving certain safety properties.

Our approach mirrors the way partial safety extends total safety. We extends the forward progress property in backward simulation to ensure that source-level error states (i.e., E_s) are preserved through the compilation, as shown below:

Definition 4.6 (Partial Forward Progress Property). Given $L_s : A_s \twoheadrightarrow B_s$, $L_t : A_t \twoheadrightarrow B_t$, $\mathbb{R}_A : A_s \Leftrightarrow A_t$, $\mathbb{R}_B : B_s \Leftrightarrow B_t$, $E_s \subseteq S_s$, $E_t \subseteq S_t$, and R, the partial forward progress property holds if:

(1), (2), and (3) are the same as those in Definition 3.4.

(4) $\forall\ w_B\ s_s\ s_t,\ (s_s, s_t) \in R(w_B) \Rightarrow s_s \in E_s \Rightarrow s_t \in \mathit{progress} \vee s_t \in E_t$.

Here, clause (4) indicates that any error state in the source (E_s) can only be compiled to either a corresponding error state in the target (E_t) or a safe state, but never to an unintended error state. We replace the original forward progress property in open backward simulation with the above definition, yielding a new backward simulation, denoted as $L_t \leqslant^{E_s,E_t}_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_s$ or $L_t \leqslant^{E}_{\mathbb{R}} L_s$. Using this definition, we establish the theorem of partial safety preservation as follows:

Theorem 4.7 (Preservation of Open Partial Safety). *Given $L_s : A_s \twoheadrightarrow B_s$, $L_t : A_t \twoheadrightarrow B_t$, $\mathbb{R}_A : A_s \Leftrightarrow A_t$, $\mathbb{R}_B : B_s \Leftrightarrow B_t$, $\mathbb{P}_{A_s}$, $\mathbb{Q}_{B_s}$ $E_s \subseteq S_s$ and $E_t \subseteq S_t$,*

$$L_s \models^{E_s} \mathbb{P}_{A_s} \twoheadrightarrow \mathbb{Q}_{B_s} \Rightarrow L_t \leqslant^{E_s,E_t}_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_s \Rightarrow L_t \models^{E_t} \mathbb{P}_{A_s} \triangleleft \mathbb{R}_A \twoheadrightarrow \mathbb{Q}_{B_s} \triangleleft \mathbb{R}_B.$$

4.3.2 Lifting Total Safety from IR to Source. We show that even when total safety is established at an intermediate stage, e.g., through safety checking on an intermediate representation (IR), it can still be propagated back to the source level. The idea is to apply Theorem 4.4 to the forward simulation (denoted as $L_s \leqslant_{\mathbb{R}} L_t$) between source and target modules. Specifically, we can reuse the existing forward simulation proofs established at each compilation pass, and additionally prove the forward progress property in the reverse direction, that is, if a target state is not stuck, then the related source state must also be not stuck. This condition ensures that the compiler does not compile an erroneous source state into a safe target state, which would otherwise make the back-propagation of total safety impossible.

Summary. We summarize the aforementioned properties of safety verification and preservation in Fig. 9, which illustrates how partial safety at the source level can derive total safety for both the target and source levels by applying them to forward (FSim) and backward (BSim) simulations, respectively. Note that we also use $L_s\ \mathbb{R}\!\geqslant\ L_t$ to represent the backward simulation $L_t \leqslant_{\mathbb{R}} L_s$.

4.4 Soundness of Open Safety

If a module M is a complete program, i.e., it has a main entry and has no external calls, then its open semantics can be transformed into closed semantics, e.g., by eliminating the transitions of X and Y. Accordingly, its open safety implies the safety under closed semantics, namely, that all reachable states can make progress. This is formalized in the following theorem:

Theorem 4.8 (Soundness of Open Safety). *Given a module M and a safety interface $\mathbb{Q}$,*

$$[\![M]\!] \models \bot \twoheadrightarrow \mathbb{Q} \Rightarrow \texttt{init}(M) \textit{ satisfies } \mathbb{Q} \Rightarrow \texttt{reach\_safe}\ [\![M]\!]_{\texttt{closed}}.$$

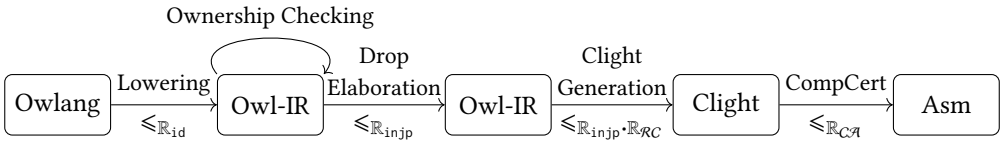


Fig. 10. Structure of the Owlang Compiler

Here, the first premise requires that M satisfies open safety under the incoming interface $\bot$, meaning that it cannot call external functions as $\bot$ cannot be satisfied. The second premise ensures that the initialization of M (e.g., memory initialization) satisfies $\mathbb{Q}$ to enable the premise for using open safety. $[\![\_]\!]_{\mathtt{closed}}$ in the conclusion represents the closed semantics.

5 Proving End-to-end Open Safety for an Ownership Language

In this section, we present the application of our open safety approach to a verified and verifying compiler for Owlang. As shown in Fig. 10, the compiler consists of a verified frontend from Owlang to Clight and a verified backend from Clight to (CompCert x86-64) assembly. The frontend proceeds in three passes. The first pass lowers Owlang into an intermediate representation called Owl-IR. The second pass performs Drop Elaboration modeled after [41], inserting explicit drop operations (like destructors in C++) for values that go out of scope. The final pass translates Owl-IR into Clight. Detailed explanations of the frontend can be found in §A. The backend is reused from the CompCertO optimizing compiler[2]. The Owlang compiler performs ownership checking at the Owl-IR level after the Lowering pass, which detects illegal uses of variables that do not have full ownership of their values.

We annotate the simulation conventions used in the proof of each pass in Fig. 10. Here, $\mathbb{R}_{\mathtt{id}}$ is the identity relation and $\mathbb{R}_{\mathtt{injp}}$ is used for the verification of compilation that changes the memory structure, e.g., by adding or removing stack allocated variables. $\mathbb{R}_{\mathtt{injp}}$ is required in Drop Elaboration as it would insert auxiliary variables to control the execution of drop operations. For the pass of Clight Generation, it is required as this pass adds C functions to implement the drop operations, which would change the structure of stack. $\mathbb{R}_{\mathcal{RC}}$ is the foreign function interface between Owlang and C, primarily relating their types and data layouts, e.g., how `enum` in Owlang is translated to the tagged union in C and how they are represented in memory. $\mathbb{R}_{\mathcal{CA}}$ is the C calling convention adopted from the compiler correctness of CompCertO [24, 55].

Compiler Correctness. We first give the correctness theorem of the Owlang compiler:

Theorem 5.1 (Compiler Correctness). *Compilation in the Owlang compiler (denoted as* `OwlComp`*) is correct in terms of open backward simulation,*

$$\forall\ (M_s : \text{Owlang})\ (M_t : \text{Asm}),\ \mathtt{OwlComp}(M_s) = M_t \Rightarrow [\![M_t]\!] \leqslant_{\mathbb{R}_{\mathcal{RA}}} [\![M_s]\!]$$

This theorem is proven by first composing the open forward simulation of the Owlang compiler frontend and the CompCertO backend, and then converting the composed forward simulation to backward simulation as CompCert does, i.e., through proving *receptiveness* for the source semantics and *determinism* for the target semantics. The result is $[\![M_t]\!] \leqslant_{\mathbb{R}_{\mathtt{id}} \cdot \mathbb{R}_{\mathtt{injp}} \cdot (\mathbb{R}_{\mathtt{injp}} \cdot \mathbb{R}_{\mathcal{RC}}) \cdot \mathbb{R}_{\mathcal{CA}}} [\![M_s]\!]$. We then refine $\mathbb{R}_{\mathtt{id}} \cdot \mathbb{R}_{\mathtt{injp}} \cdot (\mathbb{R}_{\mathtt{injp}} \cdot \mathbb{R}_{\mathcal{RC}}) \cdot \mathbb{R}_{\mathcal{CA}}$ into $\mathbb{R}_{\mathcal{RA}}$—the calling convention between Owlang and assembly, using the techniques of direct refinement [55]. As a result, $\mathbb{R}_{\mathcal{RA}}$ directly relates Owlang

[2]It supports the compilation passes from Clight to assembly except for the `Unusedglob` which removes unused module-local definitions, due to the limitation of distinguishing module-local and global definitions in CompCertO (see Section 7.3 in [55])

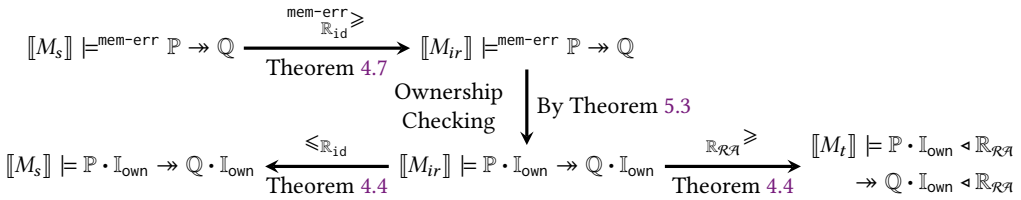


Fig. 11. Proof Structure of End-to-end Open Safety of the Owlang Compiler

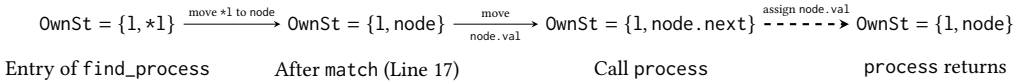


Fig. 12. The Results of Ownership Checking for `find_process`

and assembly interfaces without mentioning internal simulation conventions, which simplifies the reasoning about safety composition across languages (see §6).

End-to-end Open Safety. With ownership checking that ensures memory safety, users only need to prove partial safety of source Owlang modules. The predicate of error states in the partial safety is instantiated to `mem-err` for memory error states. We show the theorem of end-to-end safety verification and preservation of the Owlang compiler as follows:

Theorem 5.2 (End-to-end Open Safety of the Owlang Compiler). *The Owlang compiler* `OwlComp` *verifies memory safety and preserves open safety to the assembly,*

$$\forall\ (M_s : \text{Owlang})\ (M_t : \text{Asm})\ \mathbb{P}\ \mathbb{Q},\ [\![M_s]\!] \models^{\texttt{mem-err}} \mathbb{P} \twoheadrightarrow \mathbb{Q} \Rightarrow \texttt{OwlComp}(M_s) = M_t \Rightarrow [\![M_t]\!] \models \mathbb{P} \cdot \mathbb{I}_{\mathsf{own}} \triangleleft \mathbb{R}_{\mathcal{RA}} \twoheadrightarrow \mathbb{Q} \cdot \mathbb{I}_{\mathsf{own}} \triangleleft \mathbb{R}_{\mathcal{RA}} \wedge [\![M_s]\!] \models \mathbb{P} \cdot \mathbb{I}_{\mathsf{own}} \twoheadrightarrow \mathbb{Q} \cdot \mathbb{I}_{\mathsf{own}}.$$

The above conclusion consists of two parts: (1) the assembly module M_t is totally safe and (2) the source Owlang module M_s is also totally safe, under the safety interface $\mathbb{I}_{\mathsf{own}}$ imposed by the ownership checking (Definition 5.6). The proof structure of this theorem is depicted in Fig. 11, following the road map in Fig. 9. Note that $\mathbb{R}_{\mathsf{id}}$ can be absorbed by other safety interfaces and is thus omitted after the safety preservation. We preserve source-level partial safety to Owl-IR by the extended backward simulation ($\overset{\texttt{mem-err}}{\mathbb{R}_{\mathsf{id}}}\!\!\geqslant$) of the Lowering pass. This backward simulation is derived from the original forward simulation by additionally establishing clause (4) of Definition 4.6. The remaining steps follow directly from the established theorems.

5.1 Implementation and Verification of the Ownership Checking

Similar to move checking in the Rust borrow checker [42], our ownership checking is implemented as a data-flow analysis based on the Kildall algorithm from CompCert. This analysis computes, at each program point, the ownership status of variables and checks that every use of a variable is justified by ownership-—that is, only variables that hold full ownership of their values can be used.

5.1.1 Implementation. We briefly illustrate the workflow of our ownership checking using the linked list example in Fig. 2b. We present the ownership analysis results for the `find_process` function in Fig. 12, focusing on the segment from the function entry to the invocation of `process`, and then to the return from `process`. In Fig. 12, the `OwnSt` set, as computed by the analysis, represents the ownership status at each program point. It tracks the variables that have ownership of its value at that point. When a variable are moved from, then it would be removed from `OwnSt`. When a variable is assigned (or initialized), it would be added to `OwnSt`.

At the function entry of `find_process`, the analysis assumes that the input parameter `l` has full ownership of its value (i.e., the linked list). We distinguish between `l` and `*l` because our analysis tracks ownership at the level of memory blocks, i.e., whether a block owns the value stored in it. After the `match` statement, the ownership of `*l` is transferred to `node`. When calling `process`, part of `node`'s ownership, namely `node.val`, is transferred to the environment. Once the return value of `process` is assigned back to `node.val`, the full ownership of `node` is restored.

5.1.2 Verification. The goal of verifying the ownership checking is to ensure that any module that passes the ownership checking has no memory error. As introduced in §4.1.1, we formalize the correctness of verifying pass as the implication from partial safety to open safety. First, we should define the predicate for memory error state, which we denote as `mem-err`. The definition of `mem-err` utilizes the permission model in the CompCert memory. Specifically, a state belongs to `mem-err` if and only if its *next* memory access operation would violate the permission required by the accessed memory block. For example, in the CompCert memory, freeing a memory block first checks whether the block has the `Freeable` permission. If it does, the block is freed by clearing its permission; otherwise, the memory operation is undefined.

Following Definition 4.2, we prove the correctness of the ownership checking (denoted by `OwnCheck`) as shown below:

Theorem 5.3 (Correctness of the Ownership Checking). *Given an* Owl-IR *module* M, $\mathbb{P}$ *and* $\mathbb{Q}$, `OwnCheck` *is correct in terms of ensuring memory safety:*

$$[\![M]\!] \models^{\texttt{mem-err}} \mathbb{P} \twoheadrightarrow \mathbb{Q} \Rightarrow \texttt{OwnCheck}(M) \Downarrow \Rightarrow [\![M]\!] \models \mathbb{P} \cdot \mathbb{I}_{\mathsf{own}} \twoheadrightarrow \mathbb{Q} \cdot \mathbb{I}_{\mathsf{own}}.$$

Here, $\mathbb{I}_{\mathsf{own}}$ is the safety interface imposed by the ownership checking, which we refer to as *ownership interface*. The checked module is guaranteed to adhere to $\mathbb{I}_{\mathsf{own}}$ automatically, but it also relies on the environment to respect $\mathbb{I}_{\mathsf{own}}$ as well; otherwise, the memory safety guarantee may no longer hold.

High-level Proof Structure. The idea of the verification is to show that there is a safety invariant for the modules that pass the ownership checking, and this invariant has no intersection with `mem-err`. By the definition of open safety, the verification of the above theorem demands a safety invariant *Inv* for the checked module. With the partial safety premise whose safety invariant is Inv_p, we define *Inv* such that $s \in Inv \Leftrightarrow s \in Inv_p \wedge s \in I_{\mathsf{own}}$. The *ownership invariant* I_{own} relates the produced `OwnSt` (i.e., the abstract domain) and the memory state (i.e., the concrete domain). Inside the proof, the preservation of *Inv* is proved by the preservation of Inv_p from the premise of partial safety and the preservation of I_{own}. The make progress requirement is established by combining the make progress requirement of partial safety with the fact that the ownership invariant I_{own} excludes memory error states, i.e., $I_{\mathsf{own}} \cap \texttt{mem-err} = \emptyset$.

To account for interference from the environment, we define the *ownership interface* $\mathbb{I}_{\mathsf{own}}$ as an abstraction of I_{own} at module boundaries. We prove that: (1) on the outgoing side, any state satisfying I_{own} guarantees that $\mathbb{I}_{\mathsf{own}}$ holds for outgoing queries and replies; and (2) on the incoming side, any query or reply satisfying $\mathbb{I}_{\mathsf{own}}$ ensures that I_{own} can be established in the resulting state.

Ownership Invariant and Ownership Interface. We illustrate the definition of I_{own} and $\mathbb{I}_{\mathsf{own}}$ using Fig. 13, where the upper part is the transition of ownership status (the same as Fig. 12), and the lower part is the transition of memory states. The memory is represented as blocks, each of which is denoted as $b_{\mathtt{x}}$ for the location of some variable `x` (except for b_{v_1} in Fig. 13). A pointer value (b, o) contains a pair of memory block and offset which is often omitted if it is zero. We use unshaded memory blocks to indicate that the memory locations own values, and shaded blocks for those do not.

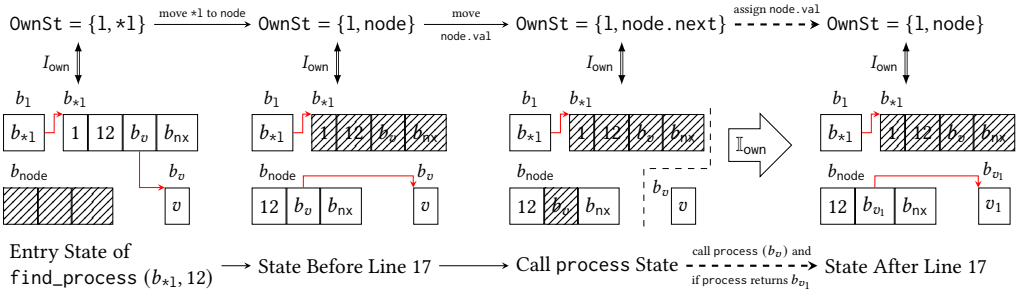


Fig. 13. Relation Between Ownership Status (Above) and Memory States (Below)

Semantic Interpretation for the Ownership Invariant. Since our goal is to prove $I_{\mathsf{own}} \cap \mathtt{mem\text{-}err} = \emptyset$ and the ownership checking has guaranteed that only variables in `OwnSt` can be used, I_{own} should require that memory locations that may be accessed by the variable in I_{own} are valid. In terms of CompCert memory, we define "valid" as: the memory location has freeable permission (i.e., the highest permission). As in ownership type systems like Rust, values are treated as resources (e.g., memory resources). Therefore, the meaning of "memory locations that may be accessed" should correspond to the memory resources owned by this variable. We can thus define I_{own} as follows:

Definition 5.4 (Ownership Invariant (Simplified)). Given `OwnSt` produced by the ownership checking and memory m, I_{own} `OwnSt` m holds if there is a resource environment Ω such that:

$$\forall\ p\ v\ fp,\ \ p \in \mathtt{OwnSt} \Rightarrow \mathrm{load}(m, p) = v \Rightarrow \Omega(p) = fp \Rightarrow \mathtt{wt\text{-}val}\ m\ fp\ v$$

The resource environment Ω maps variables to their memory resources fp (i.e., memory footprint). In details, fp is the semantic interpretation of the variable types, which is an algebraic structure that represents the memory footprint that can be touched by a value of this type. The conclusion `wt-val` m fp v requires that the value v loaded from the memory location of p is semantically well-typed. In simple terms, `wt-val` specifies that the memory blocks in fp are reachable from v and are also freeable (i.e., safe to access). For example, consider the variable `node` at the second state of Fig. 13. Since `node` is in `OwnSt` and its value is $[12; b_v; b_{\mathsf{nx}}]$, I_{own} ensures that the footprint of `node`, i.e., the blocks b_v and b_{nx} including its footprint (i.e., the rest of the list) are freeable. Therefore, the value b_v passed to the `process` is guaranteed not to be a dangling pointer.

Rely-guarantee Conditions for the Ownership Interface. At the boundary of Owlang modules, passing values to the environment can be viewed as transferring their memory footprint. To ensure that I_{own} still holds after the environment returns, we rely that the environment does not modify the memory outside the transferred footprint, i.e., memory footprint owned by the module. For instance, if the environment (i.e., `process`) frees the block b_{nx}, then I_{own} cannot be reestablished after `process` returns as `node` is in `OwnSt`. Specifically, we define a Kripke relation called `own` to protect memory that is not transferred to the environment and then use it to define $\mathbb{I}_{\mathsf{own}}$:

Definition 5.5 (Kripke Relation with Protection on Ownership Memory). $\mathtt{own} = \langle W_{\mathsf{own}}, \rightsquigarrow_{\mathsf{own}} \rangle$ where $W_{\mathsf{own}} = ((\mathtt{list\ block}) \times \mathtt{mem})$ and $\rightsquigarrow_{\mathsf{own}}$ is the accessibility relation such that

$$\begin{aligned}(fp, m) \rightsquigarrow_{\mathsf{own}} (fp', m') \Leftrightarrow\ & (\forall\ b\ \delta,\ \ b \notin fp \Rightarrow (b, \delta) \in \mathtt{unchanged\text{-}on}(m, m')) \\ & \wedge\ (\forall\ b,\ \ b \notin fp \Rightarrow b \in fp' \Rightarrow \neg\mathtt{valid\text{-}block}(m, b))\end{aligned}$$

Definition 5.6 (Ownership Interface (Simplified)). $\mathbb{I}_{\mathsf{own}} = \langle W_{\mathsf{own}}, \mathbb{I}^q_{\mathsf{own}}, \mathbb{I}^r_{\mathsf{own}} \rangle$ where

$$v_f[sg](\vec{v})@m \in \mathbb{I}^q_{\mathsf{own}}(fp, m) \Leftrightarrow \mathtt{norepeat}(fp) \wedge \overrightarrow{\mathtt{wt\text{-}val}}\ m\ fp\ \vec{v}$$

$$v@m' \in \mathbb{I}^r_{\mathsf{own}}(fp, m) \quad \Leftrightarrow \exists\, fp'\ rfp,\ (fp, m) \rightsquigarrow_{\mathsf{own}} (fp', m') \wedge rfp \subseteq fp' \wedge \texttt{wt-val}\ m'\ rfp\ v$$

Here, the Kripke world of own is a pair consisting of a list of memory blocks representing the memory footprint and the memory itself. The protection is enforced through the accessibility relation, where the unchanged-on predicate ensures that memory outside the footprint remains unchanged. Additionally, any fresh footprint (i.e., memory blocks in fp' but not in fp) is considered invalid in m as it is allocated by the environment. For the ownership interface, its domain is the Owlang interface, whose queries and replies follow the same format as the C interface, except that the signatures (sg) use Owlang types instead of C types. $\mathbb{I}^q_{\mathsf{own}}$ requires that the footprint fp contains no duplicates, and that each argument of $\vec{v}$ satisfies wt-val with its footprint in fp. $\mathbb{I}^r_{\mathsf{own}}$ protects the memory outside fp with $\rightsquigarrow_{\mathsf{own}}$ and ensures the return value satisfies wt-val.

To illustrate the definition of $\mathbb{I}_{\mathsf{own}}$, we use the transfer of memory footprint during the call to process in Fig. 13 as an example. For the outgoing query, the Kripke world of $\mathbb{I}_{\mathsf{own}}$ is set to $([b_v], m)$ where b_v is the block pointed to by node.val and m is the memory state before the call. Since I_{own} ensures that $\texttt{wt-val}\ m\ [b_v]\ (b_v, 0)$ holds (as $\texttt{node} \in \texttt{OwnSt}$ before the call), the query condition $\mathbb{I}^q_{\mathsf{own}}$ is satisfied. Upon receiving the reply $(b_{v_1}, 0)@m$, the reply condition $\mathbb{I}^r_{\mathsf{own}}$ guarantees that the memory outside fp (i.e., $[b_v]$), specifically, the region to the left of the dashed line in Fig. 13, remains unchanged. This ensures that I_{own} can be reestablished after returning from the environment.

6 End-to-end Compositional Verification of Safety for Heterogeneous Modules

We give a formal account of end-to-end safety verification for heterogeneous modules based on our approach, using the hash map example in Fig. 2. The proof structure is depicted in Fig. 14. As shown at the bottom, our **final result** is that the linked assembly program is safe under the *closed* semantics, which matches the assembly semantics in the original CompCert. For open semantics $[\![\_]\!]_\_$, we annotate their corresponding language interfaces.

Source Level Verification[(a.1) to (a.2)]. We prove partial safety for list.rs and total safety for buks.c, following Definition 4.1. For (a.1), its incoming safety interface $\mathbb{P}$ mainly specifies the assertions for hash. Since the outgoing interface of list.rs (i.e., the assertions for process) imposes no specific constraints, we denote it by $\top$ (i.e., True). We define $\mathbb{P}$ as follows:

Definition 6.1 (Safety Interface $\mathbb{P}$). $\mathbb{P} = \langle \texttt{ident} \times \texttt{symtbl} \times \texttt{int}, \mathbb{P}^q, \mathbb{P}^r \rangle$ where

$$\mathbb{P}^q(f, se, r) := \{v_f[sg]([\texttt{k}; \texttt{range}])@m \mid \texttt{find}(se, v_f) = f \wedge f = \texttt{hash} \wedge \texttt{range} > 0 \wedge r = \texttt{range}\} \cup \{v_f[sg]([\texttt{l}; \texttt{k}])@m \mid \texttt{find}(se, v_f) = f \wedge f = \texttt{find\_process}\}$$

$$\mathbb{P}^r(f, se, r) := \{v@m \mid f = \texttt{hash} \Rightarrow 0 \le v < r\}.$$

Here, the Kripke world of $\mathbb{P}$ stores the identifier of the callee, a symbol table used to find the function identifiers and an auxiliary variable r used to remember the value of range. $\mathbb{P}^q$ requires that if the callee is hash then the range must be greater than 0 and is equal to r. $\mathbb{P}^r$ requires that if the callee is hash then the return value v must be less than r (i.e., range). Inside the safety verification of list.rs, the requirement of partial safety simplifies the memory reasoning, as one can assume that all the memory operations succeed. Otherwise, the partial safety is satisfied directly.

For (a.2), we must ensure two additional conditions: one for multi-language interoperation, and the other for the interaction between safe and unsafe languages. First, the invocations of buks.c must conform to the conventions prescribed by the Owlang interface, which is ensured by $\mathbb{R}_{\mathcal{RC}}$. For example, when a C module calls an Owlang function, we must ensure that the passed values conform to the expected Owlang types, such as maintaining compatible struct layouts and field orderings between the two languages. Second, at the module boundaries, the C module must be encapsulated by the ownership interface $\mathbb{I}_{\mathsf{own}}$, which ensures that the memory safety of the Owlang module is not violated.

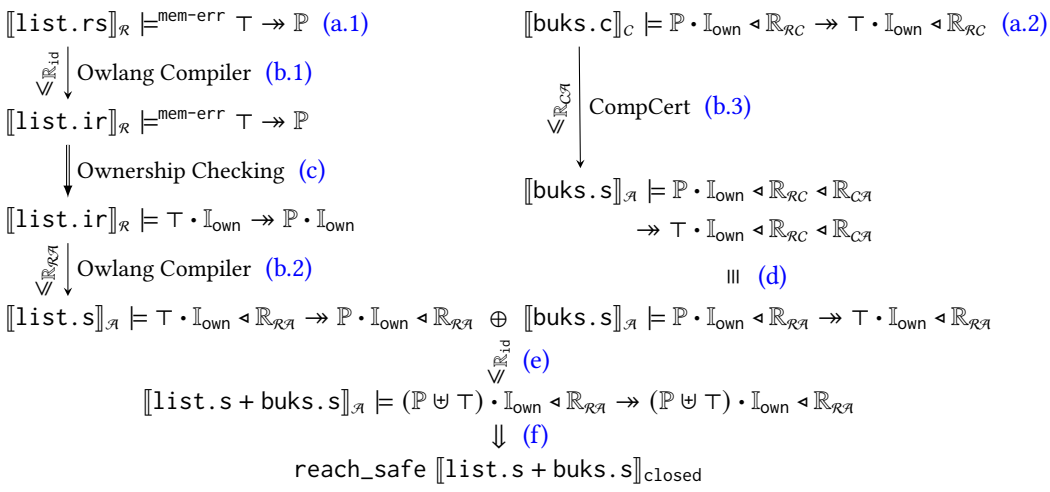


Fig. 14. Proof Structure of the Hash Map Example

Safety Preservation and Safety Checking[(b.1) to (b.3), and (c)]. We apply Theorem 5.2 to the source partial safety of `list.rs` to prove its total safety at the assembly. The generic proof structure of (b.1), (b.2), and (c) has been presented in Fig. 11, which is independent of specific examples. (b.3) is obtained by applying Theorem 4.4 to the end-to-end open backward simulation of CompCertO. Importantly, no modification to the existing proofs in CompCertO is required.

Refinement of Safety Interfaces[(d)]. To enable the target-level composition, we need to ensure that the composition of $\mathbb{R}_{\mathcal{RC}}$ (i.e., the FFI between Owlang and C) and $\mathbb{R}_{\mathcal{CA}}$ (i.e., calling convention of CompCertO) correctly reflects the calling convention $\mathbb{R}_{\mathcal{RA}}$ of the Owlang compiler. This is achieved by the refinement of safety interfaces, which derives open safety from another by strengthening its preconditions or weakening its postconditions as in program logics.

Definition 6.2. Given $\mathbb{P}$ and $\mathbb{Q}$, $\mathbb{P}$ is refined by $\mathbb{Q}$ (denoted as $\mathbb{P} \sqsubseteq \mathbb{Q}$) if:

$$\forall\ w_{\mathbb{Q}}\ q,\ q \in \mathbb{Q}^q(w_{\mathbb{Q}}) \Rightarrow \exists\ w_{\mathbb{P}},\ q \in \mathbb{P}^q(w_{\mathbb{P}}) \wedge (\forall\ r,\ r \in \mathbb{P}^r(w_{\mathbb{P}}) \Rightarrow r \in \mathbb{Q}^r(w_{\mathbb{Q}}))$$

We write $\mathbb{P} \equiv \mathbb{Q}$ if $\mathbb{P} \sqsubseteq \mathbb{Q}$ and $\mathbb{Q} \sqsubseteq \mathbb{P}$. We can strengthen (weaken) the interfaces in the open safety:

Theorem 6.3. *Given* $L : A \twoheadrightarrow B$, *if* $\mathbb{P}'_A \sqsubseteq \mathbb{P}_A, \mathbb{Q}_B \sqsubseteq \mathbb{Q}'_B$ *and* $L \models \mathbb{P}_A \twoheadrightarrow \mathbb{Q}_B$ *then* $L \models \mathbb{P}'_A \twoheadrightarrow \mathbb{Q}'_B$

In our example, the proof of (d) is derived from the application of the above theorem to $\mathbb{R}_{\mathcal{RC}} \triangleleft \mathbb{R}_{\mathcal{CA}} \equiv \mathbb{R}_{\mathcal{RA}}$. Since $\mathbb{R}_{\mathcal{RA}}$ and $\mathbb{R}_{\mathcal{CA}}$ are derived via the *transitivity* of simulation conventions [55], they directly relates $\mathcal{R}$ (respectively, C) and $\mathcal{A}$ without exposing the internal details of the compilation. Thanks to these precise calling conventions, the rewriting from $\mathbb{R}_{\mathcal{RC}} \triangleleft \mathbb{R}_{\mathcal{CA}}$ to $\mathbb{R}_{\mathcal{RA}}$ does not need to account for the differences in the compilation pipelines between the Owlang compiler and CompCertO, such as the compilation passes of the Owlang compiler frontend.

Safety Composition[(e)]. This step is proved by first applying the compositioanlity of open safety (Theorem 4.3) to the two safety judgements, and then applying the safety preservation (Theorem 4.4) to the adequacy of assembly-level linking from CompCertO, i.e., $[\![\texttt{list.s} + \texttt{buks.s}]\!]_{\mathcal{A}} \leqslant_{\mathbb{R}_{\mathrm{id}}} [\![\texttt{list.s}]\!]_{\mathcal{A}} \oplus [\![\texttt{buks.s}]\!]_{\mathcal{A}}$. Note that the safety interface $(\mathbb{P} \uplus \top) \cdot \mathbb{I}_{\mathsf{own}} \triangleleft \mathbb{R}_{\mathcal{RA}}$ in Fig. 14 is equivalent to $(\mathbb{P} \cdot \mathbb{I}_{\mathsf{own}} \triangleleft \mathbb{R}_{\mathcal{RA}}) \uplus (\top \cdot \mathbb{I}_{\mathsf{own}} \triangleleft \mathbb{R}_{\mathcal{RA}})$ that is derive directly from the safety composition.

Open Safety to Reachability Safety[(f)]. We extend `buks.c` by adding a `main` function along with hash map initialization routine (e.g., allocating the array of buckets). This ensures that the

linked program (i.e., `list.s` + `buks.s`) is a whole program. We then apply the soundness of open safety (Theorem 4.8) to establish that this whole program is reachable safe.

7 Evaluation and Related Work

Our Rocq development took about 15 person-months and 45.4k lines of code (LOC) on top of CompCertO [24, 55]. We wrote 1.7k LOC for the open safety framework as introduced in §4. Although the framework is lightweight—in the sense that it does not require modifying the existing compiler correctness proofs—we spent significant effort in identifying a suitable notion of modular safety that can be preserved through compilation, as explained in §2. In particular, we explored defining reachability-based safety on the modular semantics and attempted to prove the preservation of VST's safeN predicate, but both approaches turned out to be unsuccessful. Most of the Rocq development is spent on the Owlang compiler and the ownership checking as discussed in §5, which contain 25.3k and 11.4k LOC, respectively. Finally, we added 7.1k LOC for the safety verification of the hash map example, most of which comes from the manual proof of the source-level verification, as we need to prove that the invariant is preserved under each small-step transition. Automating this process or integrating state-of-the-art verification tools is left as future work.

CompCert-based approaches to safety preservation. Verified Software Toolchain (VST) [4, 6, 9] extends the end-to-end simulation of CompCert [29, 30], to preserve partial correctness for whole programs from Clight to the assembly. The partial correctness in VST, i.e., safeN, requires that the initial state of the module does not get stuck in any of the first N steps (with each external call counting as one). Therefore, preserving safeN for open modules requires one to show that for any execution history with N steps in the target, there is a history with M steps in the source such that they are simulated. Then one can apply the forward progress property to prove that the target can take $N + 1$ steps. However, as discussed in §2, preserving such kind of "big-step" safety through open simulation is in general not possible. Another problem for the preservation of safeN is that the *stuttering* used in the simulation [35] would make the construction of M extremely difficult, as the stutter steps are hidden (i.e., existentially quantified) in the simulation.

A recent extension of VST supports compositional verification of Rocq (Coq) and C programs [25]. They use CertiCoq [3] and CompCert to compile Rocq and C programs, respectively. The safety of C programs is verified in VST. The safety of Rocq programs is directly verified in Rocq. However, the verification results in their framework are composed at the C level via verified FFI (VeriFFI). As discussed above, VST remains limited in supporting safety preservation for open modules.

The line of reasearch on verified compositional compilation (VCC) [9, 24, 47, 55] has investigated how to establish the semantics equivalence of open modules under open-world assumptions. These works adopt small-step open simulations to ensure transitivity, which simplifies compiler interfaces. However, they primarily focus on proving the refinement between source and target modules. Building on this foundation, our work investigates how to preserve safety of open module and to support the verification of compositional compilers that perform safety checking, a problem that remains largely unexplored in existing VCC frameworks.

Safety preservation by logical relations. Prior work on compiler verification for higher-order language [8, 20] interprets types into *binary logical relations* to establish semantics equivalence between source and target modules, which states that source and target modules both diverge or both halt without failure, i.e., it implies safety preservation. However, proving the transitivity of logical relations is notoriously difficult [2, 19], which makes them challenging to be applied to realistic compilers. Moreover, these approaches often require designing a type system for each intermediate language, which deviates from our goal of reusing existing proofs in verified compilers.

Pilnser [38] is a verified multi-pass compiler for a higher-order imperative language. The semantics preservation of Pilnser is defined as *parametric inter-language simulations*, which combines bisimulation and Kripke logical relation [19] to obtain vertical transitivity. To our knowledge, the safety preservation for open module in Pilsner remains unexplored. In the Pilsner paper, the source language is equipped with a type system to ensure that well-typed programs do not go wrong. However, in order to preserve this safety property to the target level, one would need either to design type systems for all intermediate languages and rely on type-preserving compilation, or to define a form of modular safety (i.e., safety that accounts for interference from the environment) that can be preserved by their simulation techniques.

Multi-language approaches. The multi-language approach [39, 51] grows out of proof-carrying-code [36, 37] for directly verifying safety of target modules by using source-level type systems. Specifically, they interpret source types as terms written in the target language. As a result, type composition at the source level corresponds to term composition at the target level, enabling sound multi-language interoperation. In their approach, the well-typedness of a source term implies the safety of its compiled result. Although directly proving safety for the target programs is appealing, their approach incorporates the entire compiler into the semantics interpretation. That is, proving the soundness theorem must reason about the entire compilation process. It remains to be investigated that if this approach is practical for supporting multi-pass optimizing verified compilation and compositional verification involving verifying compilation.

DimSum [44] encodes calling conventions into modular semantics via *semantic wrappers* to support multi-language interoperation. We encode calling conventions into the open safety via $\triangleleft$ operator (Definition 4.5), where the $\exists$ quantifier in the pre- and post-conditions (i.e., assume and assert) plays similar roles as the dual non-determinism [7] used in the semantic wrappers. Despite this similarity, it is unclear how to define safety in their semantic model or prove the preservation of safety by their refinements, which is a direction worth exploring.

Behavior refinement. In behavior refinement, the target module refines the source module if all behaviors (i.e., event traces) of the target are included in those of the source. As a result, any safety property satisfied by the source module will also be satisfied by the target. This allows us to establish the safety of the compiled code by verifying the safety of the source module. Conditional Contextual Refinement [46] combines contextual refinement and separation logic. They transform the refinement that relies on the pre- and post-conditions into a standard contextual refinement predicate by encoding these conditions into the module semantics using dual non-determinism [7]. Although their approach shows promise in encoding a compiler's calling convention (which can be viewed as a form of pre- and post-conditions) into contextual refinement, it is unclear how to encode such calling conventions into their underlying separation logic framework.

Verification of other safety properties. The way we define memory errors is similar to the existing work of identifying the violation of safety properties, such as thread safety (i.e., absence of data races) [21, 22] and spatial memory safety [11, 34]. Therefore, we believe that our partial safety approach should work for ensuring other safety properties. For example, since data races are undefined behaviors triggered by memory writes and reads, we can extend the error state predicate to describe such undefined behaviors as another kind of memory error.

Linear languages with memory management. The L^3 [1] language is a linear language equipped with ownership (or capability) types and manual memory management via explicit drop operations. Our Owlang is an ownership subset of Rust, which is also a linear language. The type system of L^3 guarantees that each drop operation is executed if and only once. Recent work [52] further shows that such linear type systems can ensure free of memory leaks. In contrast, the memory management in Owlang follows that in the Rust language, i.e., combining the ownership

semantics and RAII (Resource Acquisition Is Initialization) to insert drop operations to release memory resource. In this aspect, our main contribution is the formal verification of the drop elaboration pass in a realistic verified compiler. One future direction is to verify that the insertion of drop operations can also guarantee free of memory leaks.

With respect to semantic type soundness for linear languages, the semantic interpretation of ownership types in L^3 divides the heap memory into two disjoint parts: one owned by the type and one representing the rest, much like the points-to assertion in separation logic. Similarly, in Owlang, we employ an internally disjointed tree structure (i.e., some kind of resource algebra) called the memory footprint (introduced in §5.1.2) to interpret ownership types semantically and to prove the correctness of ownership checking. Recent work [39] investigates safe interoperation between L^3 and a garbage-collected (GC) language. Their approach defines the conversion between types in the two languages as glue code written in the target language, capturing both static and dynamic semantics. Specifically, defining glue code as a no-op models static semantics (e.g., conversions between integer types), while defining it as some code representing the translation from GC-managed pointers to ownership pointers can model the dynamic semantics of the conversions between pointer types in two languages. In Owlang, we define the safety requirement for the language interoperation as a rely-guarantee condition called $\mathbb{I}_{\mathsf{own}}$ at the source-level semantics, which concerns only the static semantics. This kind of rely-guarantee condition can be composed with the calling conventions derived from the compiler, thereby enabling the representation of safety assertions for the assembly modules, e.g., $\mathbb{P} \cdot \mathbb{I}_{\mathsf{own}} \triangleleft \mathbb{R}_{\mathcal{RA}}$ for `list.s` in Fig. 14. To support more flexible interoperation—such as between Owlang and a GC'd language—it may be necessary to define explicit glue code that captures the dynamic semantics of interoperation, and to prove that this code satisfies the safety interface at the boundary of the Owlang side.

Formalization of Drop Elaboration. One key feature of Rust is its automatic resource management through ownership semantics and RAII. To implement this feature, the Rust compiler first inserts drop statements for out-of-scope variables whose types may own resources, and then performs drop elaboration to adjust these statements so that they are executed only when the corresponding variables actually own resources at runtime. A key challenge in verifying drop elaboration lies in defining the operational semantics for the language before drop elaboration [48]. As discussed in this GitHub issue, there are two main approaches: (1) augment the semantics with a dynamic ownership environment (e.g., runtime "magic tables" storing drop flags); or (2) let drop elaboration itself define the semantics, i.e., only provide semantics after drop elaboration. While the latter approach is favored in the Rust community—for example, Miri [32] defines MIR semantics after drop elaboration—it is unsuitable for verifying the Owlang (or even Rust) compiler. For compiler verification, we must define semantics for the surface language, where drop operations are not explicitly written but should still be executed for variables that are reassigned or go out of scope. Moreover, both move checking and borrow checking in the Rust compiler are performed *before* drop elaboration, which means their correctness must be established with respect to the surface language semantics rather than the post-elaboration semantics.

To our knowledge, the only formalization of drop elaboration so far is [12]. Unlike their work, which focuses on features only relevant to drop elaboration, we aim to verify the compilation of an ownership-based subset of Rust. This goal introduces several technical challenges, including proving the soundness of the initialization analysis used during drop elaboration and verifying the correctness of inserting stack-allocated drop flags (a particularly difficult task in the CompCert framework, since it alters the memory structure). The formalization in [12] introduces a tree-structure state for initialization analysis to minimize the number of generated drop flags and thus

reduce runtime overhead. We also plan to adopt this idea within the Kildall framework to define a similar structure for optimizing the generated code.

8 Conclusion and Future Work

We have presented open safety, a composable definition of safety for heterogeneous modules, and shown that it can be checked by verifying compilation and preserved end-to-end by verified compilation. The effectiveness of our approach is demonstrated on the verified and verifying compilation of an ownership language, and a hash map example composed of safe and unsafe code. There are two directions for the future work. One is to support borrow checking in our Owlang compiler, from which we can obtain a verified compiler for a sequential subset of Rust. The other is to connect the state-of-the-art verification tools (e.g., separation logics and model checkings) with our open safety to further extend the verification chain.

Extension of Owlang to a subset of Rust. We are currently extending Owlang with reference types and verifying a version of borrow checking based on Polonius [26]. From this verification, we aim to derive a safety interface for borrow checking, analogous to $\mathbb{I}_{\mathsf{own}}$ from the ownership checking. This interface can be viewed as a formalization of safe encapsulation—a rely-guarantee condition that characterizes the interaction between safe Rust and the unknown environment.

Connecting open safety with verification tools. To reduce the verification effort required under open safety, we can leverage existing verification frameworks such as Iris [23] and VST [4, 6]. The main challenge lies in bridging the gap between these tools and our notion of open safety. Taking Iris as an example, the semantic interpretation of Hoare triples in Iris is defined via the weakest precondition (wp). We observe that, by definition, wp must be preserved throughout execution, and states satisfying wp are guaranteed to make progress [14, 23, 31]. This naturally aligns with the definition of open safety. One potential problem may be the handling of step-indexing, which requires further investigation.

References

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L3: A linear language with locations. *Fundamenta Informaticae* 77, 4 (2007), 397–450.

[2] Amal J. Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Proc. 15th European Symposium on Programming (ESOP'06) (LNCS, Vol. 3924)*, Peter Sestoft (Ed.). Springer, Cham, 69–83. doi:10.1007/11693024_6

[3] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The third international workshop on Coq for programming languages (CoqPL)*.

[4] Andrew Appel. 2011. Verified Software Toolchain. In *Proc. 20th European Symposium on Programming (ESOP'11)*, Gilles Barthe (Ed.). LNCS, Vol. 6602. Springer, Saarbrucken, Germany, 1–17. doi:10.1007/978-3-642-19718-5_1

[5] Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *Proc. 16th IEEE Symposium on Logic in Computer Science*. 247–258.

[6] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program logics for certified compilers.* Cambridge University Press.

[7] Ralph-Johan Back and Joakim Wright. 2012. *Refinement calculus: a systematic introduction.* Springer Science & Business Media.

[8] Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 97–108. doi:10.1145/1596550.1596567

[9] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *ESOP'14*. 107–127. doi:10.1007/978-3-642-54833-8_7

[10] FBI CISA, NSA. 2023. The Case for Memory Safe Roadmaps: Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously.

[11] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*. 53–60. doi:10.1109/SecDev.2018.00015

[12] Viktor Fukala. 2024. Formalization of Rust Drop Elaboration. (2024).

[13] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan(Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*, Sriram K. Rajamani and David Walker (Eds.). ACM, New York, NY, USA, 595–608. doi:10.1145/2775051.2676975

[14] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 247 (Oct. 2023), 29 pages. doi:10.1145/3622823

[15] Professor Robert Harper. 2012. *Practical Foundations for Programming Languages*. Cambridge University Press, USA.

[16] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

[17] Tony Hoare. 2003. The verifying compiler: A grand challenge for computing research. *J. ACM* 50, 1 (Jan. 2003), 63–69. doi:10.1145/602382.602403

[18] The White House. 2024. Back to the Building Blocks: A Path toward Secure and Measurable Software.

[19] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The Marriage of Bisimulations and Kripke Logical Relations. In *Proc. 39th ACM Symposium on Principles of Programming Languages (POPL'12)*, John Field and Michael Hicks (Eds.). ACM, New York, NY, USA, 59–72. doi:10.1145/2103656.2103666

[20] Chung-Kil Hur and Derek Dreyer. 2011. A kripke logical relation between ML and assembly. *SIGPLAN Not.* 46, 1 (Jan. 2011), 133–146. doi:10.1145/1925844.1926402

[21] Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. In *Proc. 2019 ACM Conference on Programming Language Design and Implementation (PLDI'19)*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, New York, NY, USA, 111–125. doi:10.1145/3314221.3314595

[22] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. doi:10.1145/3158154

[23] 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), e20. doi:10.1017/S0956796818000151

[24] Jérémie Koenig and Zhong Shao. 2021. CompCertO: Compiling Certified Open C Components. In *Proc. 2021 ACM Conference on Programming Language Design and Implementation (PLDI'21)*. ACM, New York, NY, USA, 1095–1109. doi:10.1145/3453483.3454097

[25] Joomy Korkut, Kathrin Stark, and Andrew W. Appel. 2025. A Verified Foreign Function Interface between Coq and C. *Proc. ACM Program. Lang.* 9, POPL, Article 24 (Jan. 2025), 31 pages. doi:10.1145/3704860

[26] Rust Language. 2025. Polonius: defines the Rust borrow checker. GitHub repository. https://github.com/rust-lang/polonius

[27] Christopher League, Zhong Shao, and Valery Trifonov. 2002. Type-preserving compilation of Featherweight Java. *ACM Trans. Program. Lang. Syst.* 24, 2 (March 2002), 112–152. doi:10.1145/514952.514954

[28] Xavier Leroy. 2005–2023. The CompCert Verified Compiler. https://compcert.org/.

[29] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115. doi:10.1145/1538788.1538814

[30] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. doi:10.1007/s10817-009-9155-4

[31] 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. doi:10.1145/3632848

[32] Miri. 2025. Miri: An interpreter for Rust's mid-level intermediate representation. GitHub repository. https://github.com/rust-lang/miri

[33] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.* 21, 3 (May 1999), 527–568. doi:10.1145/319301.319345

[34] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 245–258. doi:10.1145/1542476.1542504

[35] Kedar S. Namjoshi. 1997. A simple characterization of stuttering bisimulation. In *Foundations of Software Technology and Theoretical Computer Science*, S. Ramesh and G. Sivakumar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 284–296.

[36] George Necula and Peter Lee. 1998. The Design and Implementation of a Certifying Compiler. In *Proc. 1998 ACM Conference on Programming Language Design and Implementation (PLDI'98)*. 333–344.

[37] George C. Necula. 1997. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 106–119. doi:10.1145/263699.263712

[38] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: a Compositionally Verified Compiler for a Higher-Order Imperative Language. In *Proc. 2015 ACM SIGPLAN International Conference on Functional Programming (ICFP'15)*, Kathleen Fisher and John H. Reppy (Eds.). ACM, New York, NY, USA, 166–178. doi:10.1145/2784731.2784764

[39] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic soundness for language interoperability. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 609–624. doi:10.1145/3519939.3523703

[40] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS'02*. 55–74.

[41] Rust Compiler Team. 2025. The Rust RFC Book: 0320-nonzeroing-dynamic-drop. https://rust-lang.github.io/rfcs/0320-nonzeroing-dynamic-drop.html

[42] Rust Compiler Team. 2025. Tracking moves and initialization. https://rustc-dev-guide.rust-lang.org/borrow_check/moves_and_initialization.html

[43] Rust for Linux Team. 2025. Rust for Linux. https://rust-for-linux.com/

[44] 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 (January 2023), 31 pages. doi:10.1145/3571220

[45] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (January 2020), 31 pages. doi:10.1145/3371091

[46] 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 (January 2023), 31 pages. doi:10.1145/3571232

[47] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*. ACM, New York, NY, USA, 275–287. doi:10.1145/2676726.2676985

[48] The Rust Team. 2023. Operational semantics of pre-drop-elab code? https://github.com/rust-lang/unsafe-code-guidelines/issues/391

[49] The Rust Team. 2025. The Rust Programming Language. https://rust-lang.org/

[50] Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 – Concurrency Theory*, Luís Caires and Vasco T. Vasconcelos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–271. doi:10.1007/978-3-540-74407-8_18

[51] Andrew Wagner, Zachary Eisbach, and Amal Ahmed. 2024. Realistic Realizability: Specifying ABIs You Can Count On. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 315 (Oct. 2024), 30 pages. doi:10.1145/3689755

[52] Andrew Wagner, Olek Gierczak, Brianna Marshall, John M. Li, and Amal Ahmed. 2025. From Linearity to Borrowing. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 415 (April 2025), 27 pages. doi:10.1145/3764117

[53] 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 (January 2019), 30 pages. doi:10.1145/3290375

[54] Ling Zhang, Yuting Wang, Yalun Liang, and Zhong Shao. 2025. CompCertOC: Verified Compositional Compilation of Multi-threaded Programs with Shared Stacks. *Proc. ACM Program. Lang.* 9, PLDI, Article 173 (June 2025), 24 pages. doi:10.1145/3729276

[55] Ling Zhang, Yuting Wang, Jinhua Wu, Jérémie Koenig, and Zhong Shao. 2024. Fully Composable and Adequate Verified Compilation with Direct Refinements between Open Modules. *Proc. ACM Program. Lang.* 8, POPL, Article 72 (Jan. 2024), 31 pages. doi:10.1145/3632914

A Verified Compilation of the Ownership Language

A.1 The Ownership Language

We present the syntax of Owlang in Fig. 15. From bottom to top of Fig. 15, an Owlang module (*Mods* in Fig. 15) is composed of a list of functions that are either internal functions or external functions. Note that we do not support global variable which is considered unsafe. In fact, one can use global variables in unsafe languages (e.g., C) and then compose the unsafe modules with Owlang modules. The syntax of internal and external functions is similar to the syntax of function

Identifiers	$id, l \in \text{String}$
Places	$p ::= id \mid *p \mid p.l \mid p \ as \ l$
Based Types	$\tau^B ::= \texttt{unit} \mid \texttt{bool} \mid \texttt{i32} \mid \texttt{f32} \mid \ldots$
Field Lists	$\varphi ::= [(l_1, \tau_1); ...; (l_n, \tau_n)]$
Types	$\tau ::= \tau^B \mid \texttt{Box}\ \tau \mid \texttt{struct}(id, \varphi) \mid \texttt{enum}(id, \varphi)$ $\mid (\tau_1, ..., \tau_n) \rightarrow \tau$
Constants	$c ::= () \mid \texttt{true} \mid \texttt{false} \mid n \mid \ldots$
Pure Expressions	$e^P ::= c \mid p \mid \texttt{uop}\ e \mid e_1\ \texttt{bop}\ e_2 \mid \texttt{cktag}\ p\ l$
Move Expressions	$e^M ::= \texttt{move}\ p$
Expressions	$e ::= e^P \mid e^M$
Statements	$s ::= \texttt{skip} \mid p := e \mid p := id :: l(e) \mid p := \texttt{Box}(e) \mid p := e_f(e_1, ..., e_n)$ $\mid \texttt{let}\ x : \tau = e\ \texttt{in}\ s \mid s_1; s_2 \mid \text{if}\ e^P\ \texttt{then}\ s_1\ \texttt{else}\ s_2$ $\mid \texttt{loop}\ s \mid \texttt{break} \mid \texttt{continue} \mid \texttt{return}\ p$
Variable Declarations	$dcl ::= id_1 : \tau_1, ..., id_n : \tau_n$
Internal Functions	$fd ::= \texttt{fn}\ id(dcl_1) \rightarrow \tau\{dcl_2; s\}$ $\quad (dcl_1 : \text{parameters},\ dcl_2 : \text{local variables})$
External Functions	$fe ::= \texttt{extern fn}\ id(dcl_1) \rightarrow \tau$
Functions	$f ::= fd \mid fe$
Open Modules	$Mods ::= \{(id_1, f_1), ..., (id_n, f_n)\}$

Fig. 15. Syntax of Owlang

in Rust. Inside the body of internal functions, we have assignment statement ($p := e$), function call statement ($p := e_f(e_1, ..., e_n)$) and other control flow statements, which are standard in imperative languages.

Owlang supports creating an enum object and then assigning the enum object to a place p by the statement $p := id :: l(e)$. One example is the assignment at line 22 of Fig. 2b (i.e., `*l = List::Cons(node)`), which creates a `List` object using `List::Cons` as the variant and `node` as the value of this variant, and then assigns it to `*l`. To support heap allocation, Owlang has $p := \texttt{Box}(e)$ statement, whose execution would allocate heap memory to store the value evaluated from the expression e. The assignee p must have `Box` type, which can also be seen as the ownership type in Owlang. Note that we encode the heap allocation in the syntax of Owlang instead of calling it from the standard library (i.e., calling the external function `Box::new(_)` in Rust syntax). This design choice can simplify the reasoning of the language.

In some imperative languages, such as Clight in CompCert, the l-value expressions and r-value expressions are not distinguished explicitly in the language syntax. Instead, we explicitly distinguish them in the Owlang syntax, where we use Places represent l-value expressions and Expressions for r-value expressions. The definition of Places is similar to that in MIR in the rust compiler, which contains local variable, deference, field access, and downcast ($p\ as\ l$). If a place is of type enum and represent variant l, then it can be downcast to the body of this variant using $p\ as\ l$. For example, at line 15 of Fig. 2b, the match arm `List::Cons(node)` would be desugared (in the phase of parsing

of our Owlang compiler) to `node := move (*l as Cons)`, meaning that the content of the variant `Cons` of `*l` is moved to `node`.

We classify the expressions into *pure* expressions (e^P) and *move* expressions (e^M). Pure expressions are used to express computation that does not modify the evaluation environment, such as arithmetic computation by unary operation (`uop`) or binary operation (`bop`). The expression `cktag` p l computes that if the place p (with some enum type) has variant l. For example, at line 13-15 of Fig. 2b, the match statement would be desugared into a sequence of if-else statements where each match branch would be translated to a `cktag` p l, such as `List::Nil` to `cktag *l Nil` and `List::Cons` to `cktag *l Cons`. The move expressions have only one constructor `move` p. One typical usage of move expression is to transfer the ownership of a place p_1 to the place p_2 by assignment $p_1 :=$ `move` p_2, where both p_1 and p_2 have type `Box`(τ) for some τ. The reason why we distinguish move expressions from pure expressions is that moving a place should not occur in any pure computation. For example, we should not write `move p + 3`. One way to avoid mixing move and pure expression is to check it in compilation, but for simplicity, we explicitly enforce it in the syntax. In our surface language (the language before parsing), we can mix move and pure expressions as they will be desugared by the parser.

We now turn to the design of types systems. There are based types (τ^B) including integer, float and others scalar types, whose inhabitant can be safely copied as the types with Copy trait in Rust. `Box` τ represents the ownership pointer that points to the heap memory. `struct`(id, φ) and `enum`(id, φ) are structure and enum types, both of which contain an identifier and a list of fields φ. For example, the `Node` type in Fig. 2b is defined as `struct(Node, [(key, i32); (val, Box i32); (next, Box List)])`, and the `List` type in Fig. 2b is defined as `enum(List, [(Nil, unit); (Cons, Node)])`. Note that here we do not expand `List` in the definition of `Node` type (or expand `Node` in the definition of `List` type), which seems different from the definitions in Fig. 15. This is because `List` and `Node` are mutually dependent of each other. In our Rocq development, as CompCert does, we use a *composite environment* to map the struct/enum identifiers to their field lists. So when there is any struct/enum type is used in the definition of other types, we can just refer to its type identifier rather than its full field list, e.g., we refer to `List` instead of the field list of `List` in the definition of `Node`.

A.2 Implementation of the Owlang Compiler

The compilation chain of the Owlang compiler is depicted in Fig. 10, which consists of three compilation passes that transform Owlang programs to Clight programs. We illustrate these three passes using a running example shown in Fig. 16, where the source Owlang module is shown in Fig. 16a, the Owl-IR module lowered from the source module is in Fig. 16b, and the Owl-IR module after Drop Elaboration (before Clight Generation) is in Fig. 16c. To improve the readability, the modules shown in Fig. 16 are written in Rust style, but the readers can also easily relate them to the formalized syntax defined in Fig. 15. Note that in practice, users do not need to explicitly write **`move`**, as it can be inferred by the parser.

The source program (Fig. 16a) defines a structure `Point` containing two fields of type `Box i32`. In the main function, it initializes a place `p` of type `Point`, and then it *randomly* assign (i.e., move) one of the fields of `p` to the place `a`. We want to use this program to illustrate the mechanism of *automatic memory management* in Owlang, i.e., how the resources of ownership pointers are freed, and how can it be compiled down to the Clight program, where we will see the necessity of the introduction of *ownership semantics*.

Lowering. The Lowering pass (Fig. 16b) makes variable scopes explicit and inserts `drop` statements (shown in blue) at points where ownership variables (i.e., variables whose types contain `Box`) either go out of scope or are reassigned.

```
1 struct Point {x: Box<i32>, y: Box<i32>}
2 fn test() {
3   let p = Point {x: Box(1), y: Box(2)};
4   let a: Box<i32>;
5   if rand() { a = move p.x; }
6   else { a = move p.y; }
7   return; }
```

(a) The Source Owlang Module

```
1 fn test() {
2   drop(p);
3   p = Point {x: Box(1), y: Box(2)};
4   if rand() { drop(a); a = move p.x; }
5   else { drop(a); a = move p.y; }
6   drop(a); drop(p);
7   return; }
```

(b) The Owl-IR Module (After Lowering)

```
1  fn test() {
2    flag_px = false; flag_py = false;
3    drop(p); p = Point {x: Box(1), y: Box(2)}; // Owned = {p.x, p.y}, Unowned = {a}
4    flag_px = true; flag_py = true;
5    if rand() {
6      drop(a); flag_px = false; a = move p.x; } // Owned = {a, p.y}, Unowned = {p.x}
7    else {
8      drop(a); flag_py = false; a = move p.y; } // Owned = {a, p.x}, Unowned = {p.y}
9    drop(a);  // Owned = {a, p.x, p.y}, Unowned = {p.x, p.y}
10   if flag_px { drop(p.x); } if flag_py { drop(p.y); }
11   return; }
```

(c) The Owl-IR Module (After Drop Elaboration)

Fig. 16. A Running Example for the Owlang Compiler

Drop Elaboration. The Drop Elaboration pass (Fig. 16c) refines the inserted `drop` statements by analyzing their execution behavior. It retains `drop` statements that will definitely execute, removes those that are definitely skipped, and inserts dynamic checks—using auxiliary variables—for those whose execution depends on the control flow.

To determine which variables may or may not hold ownership of their values at a given program point, the compiler performs an *initialization analysis* (init-analysis). This analysis computes two sets: `Owned` and `Unowned`, which over-approximate the variables that may (or may not) be initialized, i.e., hold ownership. When control flow merges, the analysis joins these sets via union to conservatively approximate the dynamic ownership status.

At each program point, a `drop` statement is retained if the corresponding variable is not in `Unowned` (i.e., it must have ownership of its value), and removed if it is not in `Owned` (i.e., it must not have ownership of its value). For variables that fall into the intersection of both sets (e.g., `p.x` and `p.y` at line 9 of Fig. 16c), the compiler generates auxiliary variables to track their runtime ownership status, ensuring that `drop` operations execute correctly in all branches.

Clight Generation. The Clight Generation includes translating the Owlang types to C types (e.g., translating `enum` into tagged union in C) and generating drop functions for the user-defined ownership types (e.g., the destructor for the linked list implemented in Fig. 2b).

A.3 Ownership Semantics and Verification of Drop Elaboration.

We focus on the verification of the Drop Elaboration pass and omit the proofs for other passes. Before Drop Elaboration, whether a `drop(p)` statement is executed depends on dynamically checking if `p` holds ownership of its value−−a check that is encoded in the semantics of Owl-IR, which we

refer to as *ownership semantics*. After this pass, each `drop` statement is guaranteed to be executed unconditionally. This is because Drop Elaboration refines `drop` statements and introduces auxiliary variables to track whether it holds ownership, allowing their execution to be determined statically by control flow rather than the dynamic ownership semantics.

The ownership semantics is represented by a local state called the *ownership status* in the Owl-IR semantics, denoted as the set `OwnSt`, which tracks variables that currently hold ownership. A variable is added to `OwnSt` when it is assigned and removed when it is moved or dropped. For instance, the statement `a = move b` removes `b` from `OwnSt` and adds `a` to `OwnSt`, indicating ownership transfer from `b` to `a`. In the ownership semantics, executing `drop` involves checking whether the variable is in `OwnSt` to decide whether its value should be dropped (e.g., deallocation of its memory).

The key to verifying Drop Elaboration is proving the soundness of the initialization analysis (init-analysis). Recall that init-analysis is used to statically approximate the ownership information of variables, whereas in the source program, ownership semantics captures this information dynamically. The proof follows the standard approach of abstract interpretation. We show that the init-analysis soundly over-approximates the ownership semantics by establishing a subset relation between the concrete domain (i.e., the ownership status `OwnSt`) and the abstract domain (i.e., the `Owned` and `Unowned` sets in Fig. 16c). We then prove that this relation is preserved throughout the execution, ensuring that the elaborated `drop` statements faithfully simulate the dynamic `drop` behavior in the source program.