Fully Composable and Adequate Verified Compilation with Direct Refinements between Open Modules

LING ZHANG, Shanghai Jiao Tong University, China
YUTING WANG[*], Shanghai Jiao Tong University, China
JINHUA WU, Shanghai Jiao Tong University, China
JÉRÉMIE KOENIG, Yale University, USA
ZHONG SHAO, Yale University, USA

Verified compilation of open modules (i.e., modules whose functionality depends on other modules) provides a foundation for end-to-end verification of modular programs ubiquitous in contemporary software. However, despite intensive investigation in this topic for decades, the proposed approaches are still difficult to use in practice as they rely on assumptions about the internal working of compilers which make it difficult for external users to apply the verification results. We propose an approach to verified compositional compilation without such assumptions in the setting of verifying compilation of heterogeneous modules written in first-order languages supporting global memory and pointers. Our approach is based on the memory model of CompCert and a new discovery that a Kripke relation with a notion of memory protection can serve as a uniform and composable semantic interface for the compiler passes. By absorbing the rely-guarantee conditions on memory evolution for all compiler passes into this Kripke Memory Relation and by piggybacking requirements on compiler optimizations onto it, we get compositional correctness theorems for realistic optimizing compilers as refinements that directly relate native semantics of open modules and that are ignorant of intermediate compilation processes. Such direct refinements support all the compositionality and adequacy properties essential for verified compilation of open modules. We have applied this approach to the full compilation chain of CompCert with its Clight source language and demonstrated that our compiler correctness theorem is open to composition and intuitive to use with reduced verification complexity through end-to-end verification of non-trivial heterogeneous modules that may freely invoke each other (e.g., mutually recursively).

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

Additional Key Words and Phrases: Verified Compositional Compilation, Direct Refinements, Kripke Relations

ACM Reference Format:
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 (January 2024), 31 pages. https://doi.org/10.1145/3632914

[*] Corresponding author

Authors' addresses: Ling Zhang, John Hopcroft Center for Computer Science, School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, China, ling.zhang@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; 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; Jérémie Koenig, Yale University, USA, jeremie.koenig@yale.edu; Zhong Shao, Yale University, USA, zhong.shao@yale.edu.

ACM 2475-1421/2024/1-ART72
https://doi.org/10.1145/3632914

1 INTRODUCTION

Verified compilation ensures that behaviors of source programs are faithfully transported to target code, a property desirable for end-to-end verification of software whose development involves compilation. As software is usually composed of modules independently developed and compiled, researchers have developed a wide range of techniques for *verified compositional compilation* or VCC that support modules invoking each other (i.e., open), being written in different languages (i.e., heterogeneous) and transformed by different compilers [Patterson and Ahmed 2019].

We are concerned with VCC for first-order languages with global memory states and support of pointers (e.g., see Gu et al. [2015]; Jiang et al. [2019]; Koenig and Shao [2021]; Song et al. [2020]; Stewart et al. [2015]; Wang et al. [2019]). As it stands now, the proposed approaches are inherently limited at supporting open modules (e.g. libraries) as they either deviate from the native semantics of modules or expose the semantics of intermediate representations for compilation, resulting in correctness theorems that are difficult to work with for external users. In this paper, we investigate an approach that eliminates these limitations while retaining the full benefits of VCC, i.e., obtaining correctness of compiling open modules that is *fully composable*, *adequate*, and *extensional*.

1.1 Full Compositionality and Adequacy in Verified Compilation

Correctness of compiling open modules is usually described as refinement between semantics of source and target modules. We shall write L (possibly with subscripts) to denote semantics of open modules and write $L_1 \preccurlyeq L_2$ to denote that L_1 is refined by L_2. Therefore, the compilation of any module M_2 into M_1 is correct iff $[[M_1]] \preccurlyeq [[M_2]]$ where $[[M_i]]$ denotes the semantics of M_i.

To support the most general form of VCC, it is critical that the established refinements are *fully composable*, i.e., both *horizontally and vertically composable*, and *adequate for native semantics*:

$$\begin{aligned} \textbf{Vertical Compositionality:} &\quad L_1 \preccurlyeq L_2 \Rightarrow L_2 \preccurlyeq L_3 \Rightarrow L_1 \preccurlyeq L_3 \\ \textbf{Horizontal Compositionality:} &\quad L_1 \preccurlyeq L_1' \Rightarrow L_2 \preccurlyeq L_2' \Rightarrow L_1 \oplus L_2 \preccurlyeq L_1' \oplus L_2' \\ \textbf{Adequacy for Native Semantics:} &\quad [[M_1 + M_2]] \preccurlyeq [[M_1]] \oplus [[M_2]] \end{aligned}$$

The first property states that refinements are transitive. It is essential for composing proofs for multi-pass compilers. The second property guarantees that refinements are preserved by semantic linking (denoted by $\oplus$). It is essential for composing correctness of compiling open modules (possibly through different compilers). The last one ensures that, given any modules, their semantic linking coincides with their syntactic linking (denoted by +). It ensures that linked semantics do not deviate from native semantics and is essential to propagate verified properties to final target programs.

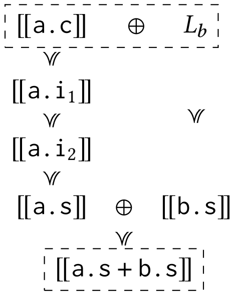
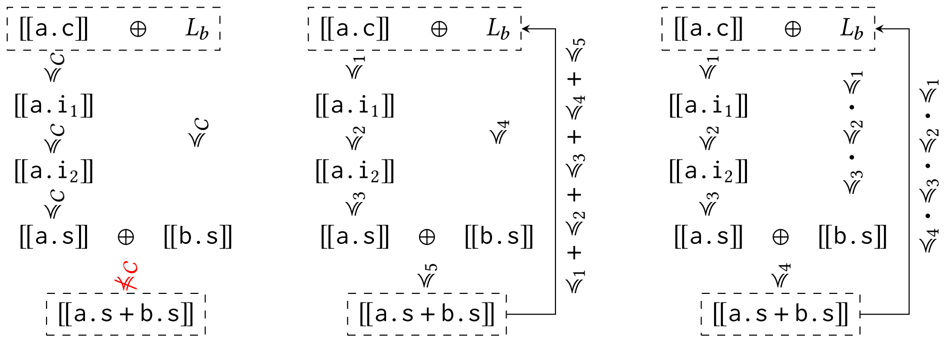


Fig. 1. Motivating Example

We use the example in Fig. 1 to illustrate the importance of the above properties in VCC where heterogeneous modules are compiled through different compilation chains and linked into a final target module. In this example, a source C module `a.c` is compiled into an assembly module `a.s` through a multi-pass optimizing compiler like CompCert: it is first compiled to `a.i`$_1$ in an intermediate representation (IR) for optimization (e.g., the RTL language of CompCert) and then to `a.i`$_2$ in another IR for code generation (e.g., the Mach language of CompCert). Finally, it is linked with a library module `b.s` which is not compiled at all (an extreme case where the compilation chain is empty). The goal is to prove that the semantics of linked target assembly `a.s + b.s` refines the combined source semantics [[`a.c`]] $\oplus L_b$ where L_b is the semantic specification of `b.s`, i.e., [[`a.s + b.s`]] $\preccurlyeq$ [[`a.c`]] $\oplus L_b$. The proof proceeds as follows:



(a) CompComp (b) CompCertM (c) CompCertO

Fig. 2. Refinements in the Existing Approaches to VCC

(1) Prove every pass respects refinement, from which $[\![\mathtt{a.i_1}]\!] \preccurlyeq [\![\mathtt{a.c}]\!]$, $[\![\mathtt{a.i_2}]\!] \preccurlyeq [\![\mathtt{a.i_1}]\!]$ and $[\![\mathtt{a.s}]\!] \preccurlyeq [\![\mathtt{a.i_2}]\!]$. Moreover, show $\mathtt{b.s}$ meets its specification, i.e., $[\![\mathtt{b.s}]\!] \preccurlyeq L_b$;
(2) By vertically composing the refinement relations for compiling $\mathtt{a.c}$, we get $[\![\mathtt{a.s}]\!] \preccurlyeq [\![\mathtt{a.c}]\!]$;
(3) By further horizontally composing with $[\![\mathtt{b.s}]\!] \preccurlyeq L_b$, we get $[\![\mathtt{a.s}]\!] \oplus [\![\mathtt{b.s}]\!] \preccurlyeq [\![\mathtt{a.c}]\!] \oplus L_b$;
(4) By adequacy for assembly and vertical composition, conclude $[\![\mathtt{a.s + b.s}]\!] \preccurlyeq [\![\mathtt{a.c}]\!] \oplus L_b$.

1.2 Problems with the Existing Approaches to Refinements

Despite the simplicity of VCC at an intuitive level, full compositionality and adequacy are surprisingly difficult to prove for any non-trivial multi-pass compiler. First and foremost, the formal definitions must take into account the facts that each intermediate representation has different semantics and each pass may imply a different refinement relation. To facilitate the discussion below, we classify different open semantics by *language interfaces* (or simply interfaces) which formalize their interaction with environments. We write $L : \mathcal{I}$ to denote that L has a language interface $\mathcal{I}$. For instance, $[\![\mathtt{a.c}]\!] : C$ denotes that the semantics of $\mathtt{a.c}$ has the interface C which only allows for interaction with environments through function calls and returns in C. Similarly, $[\![\mathtt{a.s}]\!] : \mathcal{A}$ denotes the semantics of $\mathtt{a.s}$ where $\mathcal{A}$ only allows for interaction at the assembly level. Note that the interface for a module may not match its native semantics. For example, $[\![\mathtt{a.s}]\!] : C$ asserts that $[\![\mathtt{a.s}]\!]$ actually converts assembly level calls/returns to C function calls/returns for interacting with C environments (e.g., extracting arguments from registers and memory to form an argument list for C function calls). In this case, $[\![\mathtt{a.s}]\!]$ *deviates* from the native semantics of $\mathtt{a.s}$. When the interface of $[\![M]\!]$ is not explicitly given, it is implicitly the native interface of M. We write $\preccurlyeq: \mathcal{I}_1 \Leftrightarrow \mathcal{I}_2$ to denote a refinement between two semantics with interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$. For instance, given $\preccurlyeq_{\mathrm{ac}}: \mathcal{A} \Leftrightarrow C$ that relates open semantics at the C and assembly levels, $[\![\mathtt{b.s}]\!] \preccurlyeq_{\mathrm{ac}} L_b$ asserts that $[\![\mathtt{b.s}]\!]$ is the native semantics of $\mathtt{b.s}$ and is refined by the C level specification L_b.

For VCC, it is essential that variance of open semantics and refinements does not impede compositionality and adequacy. The existing approaches achieve this by imposing *algebraic structures* on refinements. We categorize them by their algebraic structures below, and explain the problems facing them via three well-known extensions of CompCert [Leroy 2023] (the state-of-the-art verified C compiler) to support VCC, i.e., Compositional CompCert (CompComp) [Stewart et al. 2015], CompCertM [Song et al. 2020] and CompCertO [Koenig and Shao 2021].

Constant Refinement. An obvious way to account for different semantics in VCC is to force every semantics to use the same language interface $\mathcal{I}$ and a constant refinement $\preccurlyeq_{\mathcal{I}}: \mathcal{I} \Leftrightarrow \mathcal{I}$.

CompComp adopts this "one-type-fits-all" approach by having every language of CompCert to use C function calls/returns for module-level interactions and using a uniform refinement relation $\leqslant_C\colon C \Leftrightarrow C$ known as *structured simulation* [Stewart et al. 2015]. In this case, vertical and horizontal compositionality is established by proving transitivity of $\leqslant_C$ and symmetry of *rely-guarantee conditions* of $\leqslant_C$. However, because the C interface is adopted for assembly semantics, adequacy at the target level is lost, making end-to-end compiler correctness not provable as shown in Fig. 2a.

Sum of Refinements. A more relaxed approach allows users to choose language interfaces for different IRs from a finite collection $\{\mathcal{I}_1, \ldots, \mathcal{I}_m\}$ and refinements for different passes from a finite set $\{\leqslant_1, \ldots, \leqslant_n\}$ relating these interfaces, i.e., $\leqslant_i\colon \mathcal{I}_1 + \ldots + \mathcal{I}_m \Leftrightarrow \mathcal{I}_1 + \ldots + \mathcal{I}_m$. In essence, a constant refinement is split into a sum of refinements s.t. $L \leqslant_1 + \ldots + \leqslant_n L'$ holds if $L \leqslant_i L'$ for some $1 \leq i \leq n$. Then, every compiler pass can use $\leqslant_1 + \ldots + \leqslant_n$ as the uniform refinement relation, which is proven both composable and adequate under certain well-formedness constraints. Fig. 2b depicts such an example where semantics have both C and assembly interfaces (e.g., $[\![\texttt{a.s}]\!] : \mathcal{A}+C$) and the refinement relations $\leqslant_i\colon \mathcal{A} + C \Leftrightarrow \mathcal{A} + C (1 \leq i \leq 5)$ are tailored for each pass. This is the approach adopted by CompCertM [Song et al. 2020]. However, the top-level refinement $\leqslant_1 + \ldots + \leqslant_n$ is difficult to use by a third party without introducing complicated dependency on intermediate results of compilation. For example, horizontal composition with $\leqslant_1 + \ldots + \leqslant_n$ only works for modules *self-related* by all the refinements $\leqslant_i$ $(1 \leq i \leq n)$. Since $\leqslant_i$s are tailored for individual passes, they inevitably depend on the intermediate semantics used in compilation. Such dependency is only exacerbated as new languages, compilers and optimizations are introduced.

Product of Refinements. The previous approach effectively "flattens" the refinements for individual compiler passes into an end-to-end refinement. A different approach adopted by CompCertO [Koenig and Shao 2021] is to "concatenate" the refinements for individual passes into a chain of refinements by a product operation $(\_\cdot\_)$ such that $L \leqslant_1 \cdot \leqslant_2 L''$ if $L \leqslant_1 L'$ and $L' \leqslant_2 L''$ for some L'. Fig. 2c illustrates how it works. Vertical composition is simply the concatenation of refinements. For example, composing refinements for compiling `a.c` results in $[\![\texttt{a.s}]\!] \leqslant_3 \cdot \leqslant_2 \cdot \leqslant_1 [\![\texttt{a.c}]\!]$. Adequacy is trivially guaranteed with native interfaces. However, horizontal composition still depends on the intermediate semantics of compilation because of the concatenation. For example, in Fig. 2c, to horizontally compose with $[\![\texttt{a.s}]\!] \leqslant_3 \cdot \leqslant_2 \cdot \leqslant_1 [\![\texttt{a.c}]\!]$, it is necessary to show L_b refines $[\![\texttt{b.s}]\!]$ via the same product, i.e., to construct intermediate semantics bridging $\leqslant_1$, $\leqslant_2$ and $\leqslant_3$.

Summary. The existing approaches for VCC either lack adequacy because they force non-native language interfaces on semantics for open modules (e.g., CompComp) or lack compositionality that is truly extensional because they depend on intermediate semantics used in compilation (e.g., CompCertM and CompCertO). Such dependency makes their correctness theorems for compiling open modules (e.g., libraries) difficult to further compose with and incurs a high cost in verification.

1.3 Challenges for Direct Refinement of Open Modules

The ideal approach to VCC should produce refinements that directly relate the native semantics of source and target open modules without mentioning any intermediate semantics and support both vertical and horizontal composition. We shall call them *direct refinements of open modules*. For example, a direct refinement between `a.c` and `a.s` could be $\leqslant_{\mathrm{ac}}\colon \mathcal{A} \Leftrightarrow C$ s.t. $[\![\texttt{a.s}]\!] \leqslant_{\mathrm{ac}} [\![\texttt{a.c}]\!]$. It relates assembly and C without mentioning intermediate semantics, and could be further horizontally composed with $[\![\texttt{b.s}]\!] \leqslant_{\mathrm{ac}} L_b$ and vertically composed by adequacy to get $[\![\texttt{a.s} + \texttt{b.s}]\!] \leqslant_{\mathrm{ac}} [\![\texttt{a.c}]\!] \oplus L_b$. Note that even the top-level refinement is still open to horizontal and vertical composition, making direct refinements effective for supporting VCC for open modules.

The main challenge in getting direct refinements is tied to their "real" vertical composition, i.e., given any direct refinements $\preccurlyeq_1$ and $\preccurlyeq_2$, how to show $\preccurlyeq_1 \cdot \preccurlyeq_2$ is equivalent to a direct refinement $\preccurlyeq_3$. This is considered very technical and involved (see Hur et al. [2012b]; Neis et al. [2015]; Patterson and Ahmed [2019]; Song et al. [2020]) because of the difficulty in constructing *interpolating* program states for transitively relating evolving source and target states across *external calls* of open modules. This problem also manifests in proving transitivity for *logical relations* where construction of interpolating terms of higher-order types is not in general possible [Ahmed 2006]. In the setting of compiling first-order languages with global memory, all previous work avoids proving real vertical composition of direct refinements. Some produce refinement without adequacy by introducing intrusive changes to semantics to make construction of interpolating states possible. For example, CompComp instruments the semantics of languages with *effect annotations* to expose internal effects for this purpose. Some essentially restrict vertical composition to *closed* programs (e.g., CompCertM). Some leave the top-level refinement a combination of refinements that still exposes the intermediate steps of compilation (e.g., CompCertO). Finally, even if the problem of vertical composition was solved, it is not clear if the solution can support realistic optimizing compilers.

1.4 Our Contributions

In this paper, we propose an approach to direct refinements for VCC of imperative programs that addresses all of the above challenges. Our approach is based on the memory model of CompCert which supports first-order states and pointers. We show that in this memory model interpolating states for proving vertical compositionality of refinements can be constructed by exploiting the properties on memory invariants known as *memory injections*. The solution is based on a new discovery that a *Kripke relation with memory protection* can serve as a uniform and composable relation for characterizing the evolution of memory states across external calls. With this relation we successfully combined the correctness theorems of CompCert's passes into a direct refinement between C and assembly modules. We summarize our technical contributions below:

- We prove that `injp`—a Kripke Memory Relation with a notion of memory protection—is both uniform (i.e., memory transformation in every compiler pass respects this relation) and composable (i.e., transitive modulo an equivalence relation). The critical observation making this proof possible is that interpolating memory states can be constructed by exploiting memory protection *inherent* to memory injections and the *functional* nature of injections.
- Based on the above observation, we show that a direct refinement from C to assembly can be derived by composing open refinements for all of CompCert's passes starting from Clight. In particular, we show that compiler passes can use different Kripke relations sufficient for their proofs (which may be weaker than `injp`) and these relations will later be absorbed into `injp` via refinements of open semantics. Furthermore, we show that assumptions for compiler optimizations can be formalized as *semantic invariants* and, when piggybacked onto `injp`, can be transitively composed. Based on these techniques, we upgrade the proofs in CompCertO to get a direct refinement from C to assembly for the full CompCert, including all of its optimization passes. These experiments show that direct refinements can be obtained without fundamental changes to the verification framework of CompCert.
- We demonstrate the simplicity and usefulness of direct refinements by applying it to end-to-end verification of several non-trivial examples with heterogeneous modules that *mutually* invoke each other. In particular, we observe that C level refinements can be absorbed into the direct refinement of CompCert by transitivity of `injp`. Combining direct refinements with full compositionality and adequacy, we derive end-to-end refinements from high-level source specifications to syntactically linked assembly modules in a straightforward manner.

```
/* client.c */
int result;

void encrypt(int i,
     void(*p)(int*));

void process(int *r)
{
  result = *r;
}

int request(int i)
{
  encrypt(i,process);
  return i;
}
```

(a) Client in C

```
/* server.s */
key:
  .long 42
encrypt:
  // allocate frame
  Pallocframe 24 16 0
  // RSP[8] = i XOR key
  Pmov key RAX
  Pxor RAX RDI
  Pmov RDI 8(RSP)
  // call p(RSP + 8)
  Plea 8(RSP) RDI
  Pcall RSI
  // free frame
  Pfreeframe 24 16 0
  Pret
```

(b) Server in Asm

```
/* server_opt.s
 * key is an constant
 * and inlined in code */
encrypt:
  // allocate frame
  Pallocframe 24 16 0
  // RSP[8] = i XOR 42
  Pxori 42 RDI

  Pmov RDI 8(RSP)
  // call p(RSP + 8)
  Plea 8(RSP) RDI
  Pcall RSI
  // free frame
  Pfreeframe 24 16 0
  Pret
```

(c) Optimized Server

Fig. 3. An Example of Encryption Client and Server

The above developments are fully formalized in Coq based on the latest CompCertO which is in turn built on top of CompCert v3.10 (see the data-availability statement at the end of the paper for more details). While the formalisation of our approach is tied to CompCert's block-based memory model [Leroy et al. 2012], and applied to its particular chain of compilation, we present evidence in §7 that variants of `injp` could be adapted for alternate memory models for first-order languages, and that it may be extended to support new optimizations. Therefore, this work provides a promising direction for further evolving the techniques for VCC.

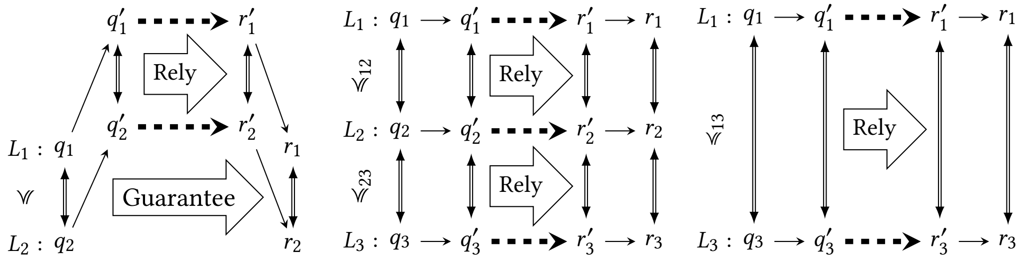
1.5 Structure of the Paper

Below we first introduce the key ideas supporting this work in §2. We then introduce necessary background and discuss the technical challenges for building direct refinements in §3. We present our technical contributions in §4, §5 and §6. We discuss the generality and limitations of our approach in §7. We discuss evaluation and related work in §8 and finally conclude in §9.

2 KEY IDEAS

We introduce a running example with heterogeneous modules and callback functions to illustrate the key ideas of our work. This example is representative of mutual dependency between modules that often appears in practice and it shows how free-form invocation between modules can be supported by our approach. As we shall see in §6, our approach also handles more complicated programs with mutually *recursive* heterogeneity without any problem.

The example is given in Fig. 3. It consists of a client written in C (Fig. 3a) and an encryption server hand-written in x86 assembly by using CompCert's assembly syntax where instruction names begin with `P` (Fig. 3b). For now, let us ignore Fig. 3c which illustrates how optimizations work in direct refinements. Users invoke `request` to initialize an encryption request. It is relayed to the function `encrypt` in the server with the prototype `void encrypt(int i, void (*p)(int*))` which respects a calling convention placing the first and second arguments in registers `RDI` and `RSI`, respectively. The main job of the server is to encrypt `i` (`RDI`) by XORing it with an encryption key (stored in the global variable `key`) and invoke the callback function `p` (`RSI`). Finally, the client


q_1'
r_1'
Rely
q_2'
r_2'
$L_1 : q_1$
r_1
$\preccurlyeq$
Guarantee
$L_2 : q_2$
r_2
$L_1 : q_1 \longrightarrow q_1'$
$r_1' \longrightarrow r_1$
$\preccurlyeq_{12}$
Rely
$L_2 : q_2 \longrightarrow q_2'$
$r_2' \longrightarrow r_2$
$\preccurlyeq_{23}$
Rely
$L_3 : q_3 \longrightarrow q_3'$
$r_3' \longrightarrow r_3$
$L_1 : q_1 \longrightarrow q_1'$
$r_1' \longrightarrow r_1$
$\preccurlyeq_{13}$
Rely
$L_3 : q_3 \longrightarrow q_3'$
$r_3' \longrightarrow r_3$

(a) Rely-Guarantee Simulation (b) Naive Vertical Composition (c) A Direct Refinement

Fig. 5. Basic Concepts of Open Simulations

takes over and stores the encrypted value in the global variable `result`. The pseudo instruction `Pallocframe m n o` allocates a stack frame of `m` bytes and stores its address in register `RSP`. In this frame, a pointer to the caller's stack frame is stored at the `o`-th byte and the return address is stored at the `n`-th byte. Note that `Pallocframe 24 16 0` in `encrypt` reserves 8 bytes on the stack from `RSP + 8` to `RSP + 16` for storing the encrypted value whose address is passed to the callback function `p`. `Pfreeframe m n o` frees the frame and restores `RSP` and the return address `RA`.

With the running example, our goal is to verify its end-to-end correctness by exploiting the direct refinement $\preccurlyeq_{\mathrm{ac}}$: $\mathcal{A} \Leftrightarrow C$ derived from CompCert's compilation chain as shown in Fig. 4. The verification proceeds as follows. First, we establish $[\![$`client.s`$]\!] \preccurlyeq_{\mathrm{ac}} [\![$`client.c`$]\!]$ by the correctness of compilation. Then, we prove $[\![$`server.s`$]\!] \preccurlyeq_{\mathrm{ac}} L_S$ manually by providing a specification L_S for the server that respects the direct refinement. At the source level, the combined semantics is further refined to a single top-level specification L_{CS}. Finally, the source and target level refinements are absorbed into the direct refinement by vertical composition and adequacy, resulting in a *single direct refinement* between the top-level specification and the target program:

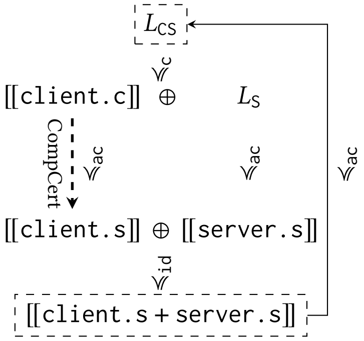


Fig. 4. Verifying the Running Example

$$[\![\texttt{client.s} + \texttt{server.s}]\!] \preccurlyeq_{\mathrm{ac}} L_{CS}$$

The refinements of open modules discussed in our paper are based on forward simulations between small-step operational semantics (often in the form of *labeled transition systems* or LTS) which have been witnessed in a wide range of verification projects [Gu et al. 2015; Jiang et al. 2019; Koenig and Shao 2021; Song et al. 2020; Stewart et al. 2015; Wang et al. 2019]. Fig. 5a depicts a refinement $L_2 \preccurlyeq L_1$ between two open semantics (LTS) L_1 and L_2. The source (target) semantics L_1 (L_2) is initialized with a query (i.e., function call) q_1 (q_2) and may invoke an external call q_1' (q_2') as the execution goes. The execution continues when q_1' (q_2') returns with a reply r_1' (r_2') and finishes with a reply r_1 (r_2). For the refinement to hold, an invariant between the source and target program states must hold throughout the execution which is denoted by the vertical double arrows in Fig. 5a. Furthermore, this refinement relies on external calls satisfying certain well-behavedness conditions (known as *rely-conditions*; e.g., external calls do not modify the private memory of callers). In turn, it guarantees the entire source and target execution satisfy some well-behavedness conditions (known as *guarantee-conditions*, e.g., they do not modify the private memory of their calling environments). The rely-guarantee conditions are essential for horizontal composition: two refinements $L_1 \preccurlyeq L_2$ and $L_1' \preccurlyeq L_2'$ with complementary rely-guarantee conditions can be composed into a single refinement $L_1 \oplus L_2 \preccurlyeq L_1' \oplus L_2'$. However, vertical composition of such refinements is difficult. A naive vertical composition of two refinements (one between L_1 and L_2 and another between L_2 and L_3) simply concatenates them together like Fig. 5b, instead of generating a single refinement between L_1 and L_3 like Fig. 5c.[1] This exposes the intermediate semantics (i.e., L_2) and imposes serious limitations on VCC as discussed in §1.2. Therefore, to the best of our knowledge, none of the existing approaches fully support the verification outlined in Fig. 4.

To address the above problem, we develop direct refinements with the following distinguishing features: *1)* they always relate the semantics of modules at their native interfaces, thereby supporting adequacy; *2)* they do not mention the intermediate process of compilation, thereby supporting heterogeneous modules and compilers; *3)* they provide direct memory protection for source and target semantics via a Kripke relation, thereby enabling horizontal composition of refinements for heterogeneous modules; *4)* most importantly, they are vertically composable. The first three features are manifested in the very definition of direct refinements, which we shall discuss in §2.1 below. We then discuss the vertical composition of direct refinements in §2.2, which relies on the discovery of the uniformity and transitivity of a Kripke relation for memory protection.

2.1 Refinement Supporting Adequacy, Heterogeneity and Horizontal Composition

To illustrate the key ideas, we use the top-level direct refinement $\preccurlyeq_{\mathsf{ac}}$ in Fig. 4 as an example. In the remaining discussions we adopt the block-based memory model of CompCert [Leroy et al. 2012] where a memory state consists of a disjoint set of *memory blocks*. $\preccurlyeq_{\mathsf{ac}}$ is a forward simulation that directly relates C and assembly modules with their native language interfaces. By the definition of these interfaces (See §3.1), a C query $q_C = v_f[sg](\vec{v})@m$ is a function call to v_f with signature sg, a list of arguments $\vec{v}$ and a memory state m; a C reply $r_C = v'@m'$ carries a return value v' and an updated memory state m'. An assembly query $q_{\mathcal{A}} = rs@m$ invokes a function with the current register set rs and memory state m. An assembly reply $r_{\mathcal{A}} = rs'@m'$ returns from a function with the updated registers rs' and memory m'. By definition, $L_2 \preccurlyeq_{\mathsf{ac}} L_1$ means that L_1 and L_2 *behave* like C and assembly programs at the boundary of modules, respectively. However, there is no restriction on how L_1 and L_2 are actually *implemented* internally, which enables source-level specifications with C interfaces like L_{S} in Fig. 4.

The rely and guarantee conditions imposed by $\preccurlyeq_{\mathsf{ac}}$ are symmetric and bundled with the simulation invariants at the boundary of modules. They make assumptions about how C and assembly queries should be related at the call sites and provide conclusions about how the replies should be related after the calls return. Given any matching source and target queries $q_C = v_f[sg](\vec{v})@m_1$ and $q_{\mathcal{A}} = rs@m_2$, it is assumed that

1. The memory states are related by an invariant j known as a *memory injection function* [Leroy et al. 2012], i.e., memory blocks in m_1 are projected by j into those in m_2;
2. The function pointer v_f is related to the program counter register in rs;
3. The source arguments $\vec{v}$ are projected either to registers in rs or to outgoing argument slots in the stack frame RSP in m_2 according to the C calling convention;
4. The outgoing arguments on the target stack frame are *freeable* and not in the image of j.

The first three requirements ensure that C arguments and memory are related to assembly registers and memory according to CompCert's C calling convention. The last one ensures outgoing arguments are protected, thereby preserving the invariant of open simulation across external calls.

[1] To simplify the presentation, we often elide the guarantee conditions in figures for simulation.

$$\begin{tikzcd}
L_S : & \texttt{encrypt}(i,p)@m_1 \arrow[r, "I_1"] \arrow[dd, leftrightarrow, "\text{initial query}"] & ([i,p],m_1) \arrow[r, "K_1"] \arrow[dd, leftrightarrow, "R"] & ([i,p],m_1') \arrow[r, "X_1"] \arrow[dd, leftrightarrow, "R"] & p(b_0)@m_1' \arrow[rr, dashed] \arrow[dd, leftrightarrow, "\text{external query}"] & & ()@m_1'' \arrow[dd, leftrightarrow, "\text{external reply}"] & \cdots \\
\preccurlyeq_{ac} & & & & & \boxed{\texttt{injp}} \Rightarrow & & \\
{[[\texttt{server.s}]]}: & rs@m_2 \arrow[r, "I_2"] & rs@m_2 \arrow[r, dashed, "K_2"] & rs'@m_2' \arrow[r, "X_2"] & rs'@m_2' \arrow[rr, dashed] & & rs''@m_2'' & \cdots
\end{tikzcd}$$

Fig. 6. Direct Refinement of the Hand-written Server

After the function calls return, the source and target queries $r_C = res@m_1'$ and $r_{\mathcal{A}} = rs'@m_2'$ must satisfy the following requirements:

(1) The updated memory states m_1' and m_2' are related by an updated memory injection j';
(2) The C-level return value res is related to the value stored in the register for return value;
(3) For any callee-saved register r, $rs'(r) = rs(r)$;
(4) The stack pointer register and program counter are restored.
(5) The access to memory during the function call is protected by a *Kripke Memory Relation* `injp` such that the private stack data for other function calls are not modified.

The first two requirements ensure that return values and memories are related according to the calling convention. The following two ensure that registers are correctly restored before returning. The last requirement plays a critical role in rely-guarantee reasoning and enables horizontal composition of direct refinements as we shall see soon.

2.1.1 Adequacy and Heterogeneity via Direct Refinement. By definition, $\preccurlyeq_{ac}$ is basically a formalized C calling convention for CompCert with direct relations between C and assembly operational semantics and with invariants for protecting register values and memory states. Adequacy is automatically guaranteed as syntactic linking coincides with semantics linking at the assembly level. That is, given any assembly modules `a.s` and `b.s`, $[[\texttt{a.s} + \texttt{b.s}]] \preccurlyeq_{id} [[\texttt{a.s}]] \oplus [[\texttt{b.s}]]$.

Moreover, $\preccurlyeq_{ac}$ does not mention anything about compilation. It works for any heterogeneous module and compilation chain that meet its requirements, even for hand-written assembly. Take the refinement of $[[\texttt{server.s}]] \preccurlyeq_{ac} L_S$ in Fig. 4 as an example. The first few steps of the simulation are depicted in Fig. 6, where L_S is an LTS hand-written by us and $[[\texttt{server.s}]]$ is derived from the CompCert assembly semantics. Because L_S is only required to respect the C interface, we choose a form easy to comprehend where its internal executions are in big steps. Now, suppose the environment calls `encrypt` with source and target queries initially related by CompCert's calling convention s.t. $rs(\texttt{RDI}) = i$ and $rs(\texttt{RSI}) = p$. After the initialization I_1 and I_2, the execution enters internal states related by an invariant R. Then, the target execution takes internal steps K_2 until reaching an external call. This corresponds to executing lines 5-13 in Fig. 3b, which allocates the stack frame `RSP`, performs encryption by storing `i XOR key` at the address `RSP+8`, and calls back p with `RSP+8`. At the source level, these steps correspond to one big-step execution K_1 which allocates a memory block b_0, stores `i XOR key` at b_0, and prepares to call p with b_0. Therefore, the memory injection in R maps b_0 to `RSP+8`. The source and target execution continue with transitions X_1 and X_2 to the external calls to p, return from p and go on until they return from `encrypt`.

2.1.2 Horizontal Composition via Kripke Memory Relations. The Kripke Memory Relation (KMR) `injp` provides essential protection for private values on the stack, which ensures that simulations between heterogeneous modules can be established and their horizontal composition is feasible.

We illustrate these points via our running example. Assume that the environment calls `request` in the client with 11 which in turn calls `encrypt` in the server to get the value 11 `XOR` 42 = 33

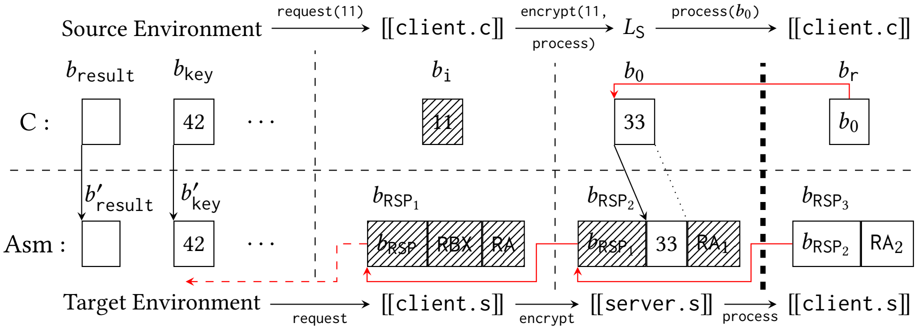


Fig. 7. Snapshot of the Memory State after Call Back

whose address is passed back to the client by calling `process`. Fig. 7 depicts a snapshot of the memory states and the injection right after `process` is entered (i.e., at line 8 in Fig. 3a), where boxes denote allocated memory blocks, black arrows between blocks denote injections, and red arrows denote pointers. The source semantics allocates one block for each local variable (b_i for `i`, b_0 for the encrypted value 33 and b_r for `r`) while the target semantics stores their values in registers or stacks (11 is stored in RDI while 33 on the stack because its address is taken and may be modified by the callee). One stack frame is allocated for each function call which stores private data including pointers to previous frames (b_{RSP}), return addresses (RA), and callee-saved registers (e.g., RBX).

`injp` is essential for proving simulation for open modules as it guarantees simulation can be re-established after external calls return. Informally, at every external call site, `injp` marks all memory regions outside the footprint (domain and image) of the current injection as *private* and does not allow the external call to modify those memory regions. From the perspective of `server.s`, when the snapshot in Fig. 7 is taken, the execution is inside the thick dashed line in Fig. 6 and protected by `injp`. Therefore, all the shaded memory in Fig. 7 are marked as private and protected against the callback to `process`. Indeed, they correspond to either memory values turned into temporary variables (e.g., b_i) or private stack data (e.g., b_{RSP}, RBX and RA in block b_{RSP_1}) that should not be touched by `process`. Such protection ensures that when `process` returns, all the private values are still valid, thereby re-establishing the simulation invariant.

The role of `injp` is reversed for the incoming calls from the environment: it guarantees that the entire execution from the initial query to the final reply will not touch any private memory of the environment. Therefore, `injp` is used to impose a reliance on memory protection by external calls and to provide a *symmetric* guarantee of memory protection for the environment callers. Any simulations with compatible language interfaces that satisfy this rely-guarantee condition can be horizontally composed. For example, we can horizontally compose [[`client.s`]] $\preccurlyeq_{\mathsf{ac}}$ [[`client.c`]] and [[`server.s`]] $\preccurlyeq_{\mathsf{ac}}$ L_S into [[`client.s`]] $\oplus$ [[`server.s`]] $\preccurlyeq_{\mathsf{ac}}$ [[`client.c`]] $\oplus$ L_S in Fig. 4.

2.2 Uniform and Transitive KMR for Vertical Composition of Direct Refinements

Direct refinements are only useful if they can be vertically composed, which is critical for composing refinements obtained from individual compiler passes into a single top-level refinement such as $\preccurlyeq_{\mathsf{ac}}$ and for further composition with source-level refinements as shown in Fig. 4.

We discuss our approach for addressing this problem by using CompCert and CompCertO as the concrete platforms. It is based on the following two observations. First, `injp` in fact captures the rely-guarantee conditions for memory protection needed by every compiler pass in CompCert. At a high-level, it means that the rely-guarantee conditions as depicted in Fig. 5 can all be replaced by `injp` (modulo the details on language interfaces). Second, `injp` is transitively composable, i.e., any vertical pairing of `injp` can be proved equivalent to a single `injp`. It means that given two refinements $L_2 \preccurlyeq_{12} L_1$ and $L_3 \preccurlyeq_{23} L_2$ as depicted in Fig. 5b, when their rely-guarantee conditions are uniformly represented by `injp`, they can be merged into the direct refinement $L_3 \preccurlyeq_{13} L_1$ in Fig. 5c with a single `injp` as the rely-guarantee condition. We shall present the technical challenges leading to these observations in §3 and elaborate on the observations themselves in §4.

By the above observations, an obvious approach for applying direct refinements to realistic optimizing compilers is to prove open simulation for every compiler pass using `injp`, and vertically compose those simulations into a single simulation. However, for a non-trivial compiler like CompCert, it means we need to rewrite a significant part of its proofs. More importantly, optimization passes in CompCert need additional rely-guarantee conditions as they are based on value analysis. To address the first problem, we start from the refinement proofs with least restrictive KMRs for individual passes in CompCertO [Koenig and Shao 2021], and exploit the properties that these KMRs can eventually be "absorbed" into `injp` in vertical composition to generate a direct refinement parameterized only by `injp`. To address the second problem, we propose a notion of *semantic invariant* that captures the rely-guarantee conditions for value analysis. When piggybacked onto `injp`, this semantic invariant can be transitively composed along with `injp` and eventually pushed to the C level. It then becomes a condition for enabling optimizations at the source level, e.g., for supporting the refinement of the optimized server in Fig. 3c. We discuss those solutions in §5.

Finally, we observe that source-level refinements can also be parameterized by `injp`, which enables end-to-end program verification as depicted in Fig. 4 as we shall discuss in §6.

3 BACKGROUND AND CHALLENGES

3.1 Background

We introduce the memory model, open simulations, and `injp` which is critical for direct refinements.

3.1.1 Block-based Memory Model. By Leroy et al. [2012], a memory state m (of type `mem`) consists of a disjoint set of *memory blocks*. A memory address or pointer (b, o) points to the o-th byte in the block b where b has type `block` and o has type `Z` (integers). The value at (b, o) is denoted by $m[b, o]$. Values (of type `val`) are either undefined (`Vundef`), 32- or 64-bit integers or floats, or pointers of the form $\texttt{Vptr}(b, o)$. For simplicity, we often write b for $\texttt{Vptr}(b, 0)$. The memory operations including allocation, free, read and write are provided and governed by permissions. The permission of a memory cell is ordered from high to low as $\texttt{Freeable} \geqslant \texttt{Writable} \geqslant \texttt{Readable} \geqslant \texttt{NA}$ where `Freeable` enables all operations, `Writable` enables all but free, `Readable` enables only read, and `NA` enables none. If $p_1 \geqslant p_2$ then any cell with permission p_1 also has permission p_2. $\texttt{perm}(m, p)$ denotes the set of cells with at least permission p. For example, $(b, o) \in \texttt{perm}(m, \texttt{Readable})$ iff the cell at (b, o) in m is readable. An address with no permission at all is not in the footprint of memory.

Transformations of memory states are captured via partial functions $j : \texttt{block} \to \lfloor \texttt{block} \times \texttt{Z} \rfloor$ called *injection functions*, s.t. $j(b) = \emptyset$ if b is removed from memory and $j(b) = \lfloor (b', o) \rfloor$ if b is shifted (injected) to (b', o) in the target memory. We define $\texttt{meminj} = \texttt{block} \to \lfloor \texttt{block} \times \texttt{Z} \rfloor$. v_1 and v_2 are related under j (denoted by $v_1 \hookrightarrow_v^j v_2$) if either v_1 is `Vundef`, or they are both equal scalar values, or pointers shifted according to j, i.e., $v_1 = \texttt{Vptr}(b, o)$, $j(b) = \lfloor (b', o') \rfloor$ and $v_2 = \texttt{Vptr}(b', o + o')$.

Given this relation, there is a *memory injection* between the source memory state m_1 and the target state m_2 under j (denoted by $m_1 \hookrightarrow_m^j m_2$) if the following properties are satisfied which ensure preservation of permissions and values under injection:

$$\forall\ b_1\ b_2\ o\ o'\ p,\ \ j(b_1) = \lfloor (b_2, o') \rfloor \Rightarrow (b_1, o) \in \texttt{perm}(m_1, p) \Rightarrow (b_2, o + o') \in \texttt{perm}(m_2, p).$$
$$\forall\ b_1\ b_2\ o\ o',\ \ j(b_1) = \lfloor (b_2, o') \rfloor \Rightarrow (b_1, o) \in \texttt{perm}(m_1, \texttt{Readable}) \Rightarrow m_1[b_1, o] \hookrightarrow_v^j m_2[b_2, o + o'].$$

Memory injections are *transitive* and necessary for verifying transformations of memory structures (e.g., merging local variables into stack-allocated data and generating a concrete stack frame). For the remaining passes, a simpler relation called *memory extension* is used instead, which employs an identity injection. Reasoning about permissions under refinements is a major source of complexity.

3.1.2 A Framework for Open Simulations. In CompCertO [Koenig and Shao 2021], a *language interface* $A = \langle A^q, A^r \rangle$ is a pair of sets A^q and A^r denoting acceptable queries and replies for open modules, respectively. Different interfaces may be used for different languages. The relevant ones for our discussion have been introduced in §2.1 and formally defined as follows. The language interface at the C level is $C = \langle \texttt{val} \times \texttt{sig} \times \texttt{val}^* \times \texttt{mem}, \texttt{val} \times \texttt{mem} \rangle$ where its queries and replies take the forms $v_f[sg](\vec{v})@m$ and $v'@m'$, respectively. The language interface at the assembly level is $\mathcal{A} = \langle \texttt{regset} \times \texttt{mem}, \texttt{regset} \times \texttt{mem} \rangle$ where its queries and replies take the form $rs@m$.

Open labeled transition systems (LTS) represent semantics of modules that may accept queries and provide replies at the *incoming side* and provide queries and accept replies at the *outgoing side* (i.e., calling external functions). An open LTS $L : A \twoheadrightarrow B$ is a tuple $\langle D, S, I, \rightarrow, F, X, Y \rangle$ where A (B) is the language interface for outgoing (incoming) queries and replies, $D \subseteq B^q$ a set of initial queries, S a set of internal states, $I \subseteq D \times S$ ($F \subseteq S \times B^r$) transition relations for incoming queries (replies), $X \subseteq S \times A^q$ ($Y \subseteq S \times A^r \times S$) transitions for outgoing queries (replies), and $\rightarrow \subseteq S \times \mathcal{E}^* \times S$ internal transitions emitting events of type $\mathcal{E}$. Note that $(s, q^O) \in X$ iff an outgoing query q^O happens at s; $(s, r^O, s') \in Y$ iff after q^O returns with r^O the execution continues with an updated state s'.

Kripke relations are used to describe evolution of program states in open simulations between LTSs. A Kripke relation $R : W \longrightarrow \{S \mid S \subseteq A \times B\}$ is a family of relations indexed by a *Kripke world* W; for simplicity, we define $\mathcal{K}_W(A, B) = W \longrightarrow \{S \mid S \subseteq A \times B\}$. A *simulation convention* relating two language interfaces A_1 and A_2 is 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 $\mathbb{R} : A_1 \Leftrightarrow A_2$. Simulation conventions serve as interfaces of open simulations by relating source and target language interfaces. For example, a C-level convention $\texttt{c} : C \Leftrightarrow C = \langle \texttt{meminj}, \mathbb{R}_{\texttt{c}}^q, \mathbb{R}_{\texttt{c}}^r \rangle$ relates C queries and replies as follows, where the Kripke world consists of injections and, in a given world j, the values and memory in queries and replies are related by j.

$$\begin{array}{lcl} (v_f[sg](\vec{v})@m, v'_f[sg](\vec{v'})@m') \in \mathbb{R}_{\texttt{c}}^q(j) & \Leftrightarrow & v_f \hookrightarrow_v^j v'_f \wedge \vec{v} \hookrightarrow_v^j \vec{v'} \wedge m \hookrightarrow_m^j m' \\ (v@m, v'@m') \in \mathbb{R}_{\texttt{c}}^r(j) & \Leftrightarrow & v \hookrightarrow_v^j v' \wedge m \hookrightarrow_m^j m' \end{array}$$

Open forward simulations describe refinement between LTS. To establish an open (forward) simulation between $L_1 : A_1 \twoheadrightarrow B_1$ and $L_2 : A_2 \twoheadrightarrow B_2$, one needs to find two simulation conventions $\mathbb{R}_A : A_1 \Leftrightarrow A_2$ and $\mathbb{R}_B : B_1 \Leftrightarrow B_2$ that connect queries and replies at the outgoing and incoming sides, and show the internal execution steps and external interactions of open modules are related by an invariant R. This simulation is denoted by $L_1 \leqslant_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$ and formally defined as follows (for simplicity, we shall write $L_1 \leqslant_{\mathbb{R}} L_2$ to denote $L_1 \leqslant_{\mathbb{R} \twoheadrightarrow \mathbb{R}} L_2$):

Definition 3.1. Given $L_1 : A_1 \twoheadrightarrow B_1$, $L_2 : A_2 \twoheadrightarrow B_2$, $\mathbb{R}_A : A_1 \Leftrightarrow A_2$ and $\mathbb{R}_B : B_1 \Leftrightarrow B_2$, $L_1 \leqslant_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$ holds if there is some Kripke relation $R \in \mathcal{K}_{W_B}(S_1, S_2)$ that satisfies:

(1) $\forall\, q_1\, q_2,\ (q_1, q_2) \in \mathbb{R}_B^q(w_B) \Rightarrow (q_1 \in D_1 \Leftrightarrow q_2 \in D_2)$
(2) $\forall\, w_B\, q_1\, q_2\, s_1,\ (q_1, q_2) \in \mathbb{R}_B^q(w_B) \Rightarrow (q_1, s_1) \in I_1 \Rightarrow \exists\, s_2, (s_1, s_2) \in R(w_B) \wedge (q_2, s_2) \in I_2.$
(3) $\forall\, w_B\, s_1\, s_2\, t,\ (s_1, s_2) \in R(w_B) \Rightarrow s_1 \xrightarrow{t} s'_1 \Rightarrow \exists\, s'_2, (s'_1, s'_2) \in R(w_B) \wedge s_2 \xrightarrow{t}{}^{*} s'_2.$
(4) $\forall\, w_B\, s_1\, s_2\, q_1,\ (s_1, s_2) \in R(w_B) \Rightarrow (s_1, q_1) \in X_1 \Rightarrow$
$\quad \exists w_A\, q_2,\ (q_1, q_2) \in \mathbb{R}_A^q(w_A) \wedge (s_2, q_2) \in X_2 \wedge$
$\quad\quad \forall\, r_1\, r_2\, s'_1, (r_1, r_2) \in \mathbb{R}_A^r(w_A) \Rightarrow (s_1, r_1, s'_1) \in Y_1 \Rightarrow \exists\, s'_2, (s'_1, s'_2) \in R(w_B) \wedge (s_2, r_2, s'_2) \in Y_2.$
(5) $\forall\, w_B\, s_1\, s_2\, r_1,\ (s_1, s_2) \in R(w_B) \Rightarrow (s_1, r_1) \in F_1 \Rightarrow \exists\, r_2, (r_1, r_2) \in \mathbb{R}_B^r(w_B) \wedge (s_2, r_2) \in F_2.$

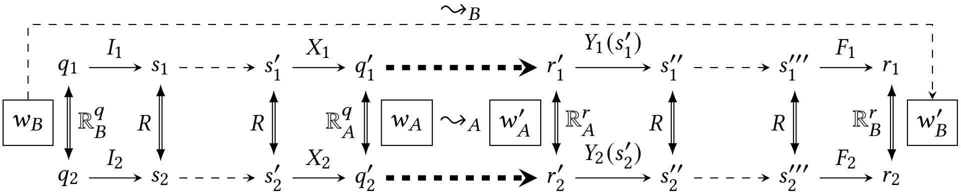


Fig. 8. Open Simulation between LTS

Here, property (1) requires initial queries to match; (2) requires initial states to hold under the invariant R; (3) requires internal execution to preserve R; (4) requires R to be preserved across external calls, and (5) requires final replies to match. According to these properties, a complete forward simulation looks like Fig. 8. From the above definition, it is easy to prove the horizontal and vertical compositionality of open simulations and adequacy for assembly modules, i.e., $\forall\ L_1\ L_2\ L'_1\ L'_2,\ L_1 \leqslant_{\mathbb{R}} L_2 \Rightarrow L'_1 \leqslant_{\mathbb{R}} L'_2 \Rightarrow L_1 \oplus L'_1 \leqslant_{\mathbb{R}} L_2 \oplus L'_2$ and $\forall\ (M_1\ M_2 : \mathsf{Asm}),\ [[M_1]] \oplus [[M_2]] \leqslant_{\mathtt{id}} [[M_1 + M_2]]$.

The Kripke worlds (e.g., memory injections) may evolve as the execution goes on. *Rely-guarantee* reasoning about such evolution is essential for horizontal composition of simulations. For illustration, the Kripke worlds at the boundary of modules are displayed in Fig. 8. The evolution of worlds across external calls is governed by an *accessibility relation* $w_A \leadsto_A w'_A$ for describing the *rely-condition*. By assuming $w_A \leadsto_A w'_A$, one needs to prove the *guarantee condition* $w_B \leadsto_B w'_B$, i.e., the evolution of worlds in the whole execution respects $\leadsto_B$. Simulations with symmetric rely-guarantee conditions can be horizontally composed, even with mutual calls between modules.

Note that the accessibility relation and evolution of Kripke worlds are not explicit in the definition of simulation conventions. Instead, they are implicit by assuming a modality operator $\Diamond$ is always applied to $\mathbb{R}^r$ s.t. $r \in \Diamond\mathbb{R}^r(w) \Leftrightarrow \exists\ w', w \leadsto w' \wedge r \in \mathbb{R}^r(w')$. We often ignore accessibility and modality when talking *purely* about simulation conventions in the remaining discussion.

Accessibility relations are mainly for describing evolution of memory states across external calls. For this, simulation conventions are parameterized by *Kripke Memory Relations* or KMR.

Definition 3.2. A Kripke Memory Relation is a tuple $\langle W, f, \leadsto, R\rangle$ where W is a set of worlds, $f : W \longrightarrow \mathtt{meminj}$ a function for extracting injections from worlds, $\leadsto \subseteq W \times W$ an accessibility relation between worlds and $R : \mathcal{K}_W(\mathtt{mem}, \mathtt{mem})$ a Kripke relation over memory states that is compatible with the memory operations. We write $w \leadsto w'$ for $(w, w') \in \leadsto$.

We write $\mathbb{R}_K$ to emphasize that a simulation convention $\mathbb{R}$ is parameterized by the KMR K, meaning $\mathbb{R}_K$ shares the same type of worlds with K and inherits its accessibility relation.

The most interesting KMR is injp as it provides protection on memory w.r.t. injections.

Definition 3.3 (Kripke Relation with Memory Protection). $\mathtt{injp} = \langle W_{\mathtt{injp}}, f_{\mathtt{injp}}, \leadsto_{\mathtt{injp}}, R_{\mathtt{injp}}\rangle$ where $W_{\mathtt{injp}} = (\mathtt{meminj} \times \mathtt{mem} \times \mathtt{mem})$, $f_{\mathtt{injp}}(j, \_, \_) = j$, $(m_1, m_2) \in R_{\mathtt{injp}}(j, m_1, m_2) \Leftrightarrow m_1 \hookrightarrow^j_m m_2$ and

$$\begin{aligned}(j, m_1, m_2) \leadsto_{\mathtt{injp}} (j', m'_1, m'_2) \ \Leftrightarrow\ & j \subseteq j' \wedge \mathtt{unmapped}(j) \subseteq \mathtt{unchanged\text{-}on}(m_1, m'_1)\\ & \wedge \mathtt{out\text{-}of\text{-}reach}(j, m_1) \subseteq \mathtt{unchanged\text{-}on}(m_2, m'_2).\\ & \wedge \mathtt{mem\text{-}acc}(m_1, m'_1) \wedge\ \mathtt{mem\text{-}acc}(m_2, m'_2)\end{aligned}$$

Here, $\mathtt{mem\text{-}acc}(m, m')$ denotes monotonicity of memory states such as valid blocks can only increase and read-only data does not change in value. $\mathtt{unchanged\text{-}on}(m, m')$ denotes memory cells whose permissions and values are not changed from m to m' and

$$\begin{aligned}(b_1, o_1) \in \mathtt{unmapped}(j) \quad & \Leftrightarrow\ j(b_1) = \emptyset\\ (b_2, o_2) \in \mathtt{out\text{-}of\text{-}reach}(j, m_1) \ & \Leftrightarrow\ \forall\ b_1\ o'_2,\ j(b_1) = \lfloor(b_2, o'_2)\rfloor \Rightarrow (b_1, o_2 - o'_2) \notin \mathtt{perm}(m_1, \mathsf{NA}).\end{aligned}$$

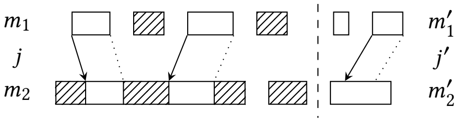


Fig. 9. Kripke Worlds Related by `injp`

By definition, a world (j, m_1, m_2) evolves to (j', m_1', m_2') under `injp` only if j' is strictly larger than j and any memory cells in m_1 and m_2 not in the domain (i.e., unmapped by j) or image of j (i.e., out-of-reach by j from m_1) will be protected, meaning their values and permissions are unchanged from m_1 (m_2) to m_1' (m_2'). An example is shown in Fig. 9 where the shaded regions in m_1 are unmapped by j and unchanged while those in m_2 are out-of-reach from j and unchanged. m_1' and m_2' may contain newly allocated blocks which are not protected by `injp`. When `injp` is used at the outgoing side, it denotes that the simulation relies on knowing that the unmapped and out-of-reach regions at the call side are not modified by external calls. When `injp` is used at the incoming side, it denotes that the simulation guarantees such regions at initial queries are not modified by the simulation itself.

3.2 Challenges for Vertically Composing Open Simulations

As discussed in §2.2, the challenge for constructing direct refinements for multi-pass optimizing compilers lies in their vertical composition. The most basic vertical composition for open simulations is stated below which is easily proved by pairing of individual simulations [Koenig and Shao 2021].

Theorem 3.4 (V. Comp). *Given $L_1 : A_1 \twoheadrightarrow B_1$, $L_2 : A_2 \twoheadrightarrow B_2$ and $L_3 : A_3 \twoheadrightarrow B_3$, and given $\mathbb{R}_{12} : A_1 \Leftrightarrow A_2$, $\mathbb{S}_{12} : B_1 \Leftrightarrow B_2$, $\mathbb{R}_{23} : A_2 \Leftrightarrow A_3$ and $\mathbb{S}_{23} : B_2 \Leftrightarrow B_3$,*

$$L_1 \leqslant_{\mathbb{R}_{12} \twoheadrightarrow \mathbb{S}_{12}} L_2 \Rightarrow L_2 \leqslant_{\mathbb{R}_{23} \twoheadrightarrow \mathbb{S}_{23}} L_3 \Rightarrow L_1 \leqslant_{\mathbb{R}_{12} \cdot \mathbb{R}_{23} \twoheadrightarrow \mathbb{S}_{12} \cdot \mathbb{S}_{23}} L_3.$$

Here, $(\_ \cdot \_)$ is a composed simulation convention s.t. $\mathbb{R} \cdot \mathbb{S} = \langle W_{\mathbb{R}} \times W_{\mathbb{S}}, \mathbb{R}^q \cdot \mathbb{S}^q, \mathbb{R}^r \cdot \mathbb{S}^r \rangle$ where for any q_1 and q_3, $(q_1, q_3) \in \mathbb{R}^q \cdot \mathbb{S}^q(w_{\mathbb{R}}, w_{\mathbb{S}}) \Leftrightarrow \exists q_2, (q_1, q_2) \in \mathbb{R}^q(w_{\mathbb{R}}) \wedge (q_2, q_3) \in \mathbb{S}^q(w_{\mathbb{S}})$ (similarly for $\mathbb{R}^r \cdot \mathbb{S}^r$). Then, given any compiler with N passes and their refinement relations $L_1 \leqslant_{\mathbb{R}_{12} \twoheadrightarrow \mathbb{S}_{12}} L_2, \ldots, L_N \leqslant_{\mathbb{R}_{N,N+1} \twoheadrightarrow \mathbb{S}_{N,N+1}} L_{N+1}$, we get their concatenation $L_1 \leqslant_{\mathbb{R}_{12} \cdot \ldots \cdot \mathbb{R}_{N,N+1} \twoheadrightarrow \mathbb{S}_{12} \cdot \ldots \cdot \mathbb{S}_{N,N+1}} L_{N+1}$, which exposes internal compilation and weakens compositionality as we have discussed in §1.2.

The above problem may be solved if the composed simulation convention can be *refined* into a single convention directly relating source and target queries and replies. Given two simulation conventions $\mathbb{R}, \mathbb{S} : A_1 \Leftrightarrow A_2$, $\mathbb{R}$ is *refined* by $\mathbb{S}$ if

$$\begin{aligned} \forall\, w_{\mathbb{S}}\ q_1\ q_2,\ (q_1, q_2) \in \mathbb{S}^q(w_{\mathbb{S}}) \Rightarrow \exists\, w_{\mathbb{R}},\ (q_1, q_2) \in \mathbb{R}^q(w_{\mathbb{R}}) \wedge \\ \forall\, r_1\ r_2,\ (r_1, r_2) \in \mathbb{R}^r(w_{\mathbb{R}}) \Rightarrow (r_1, r_2) \in \mathbb{S}^r(w_{\mathbb{S}}) \end{aligned}$$

which we write as $\mathbb{R} \sqsubseteq \mathbb{S}$. If both $\mathbb{R} \sqsubseteq \mathbb{S}$ and $\mathbb{S} \sqsubseteq \mathbb{R}$, then $\mathbb{R}$ and $\mathbb{S}$ are equivalent and written as $\mathbb{R} \equiv \mathbb{S}$. By definition, $\mathbb{R} \sqsubseteq \mathbb{S}$ indicates any query for $\mathbb{S}$ can be converted into a query for $\mathbb{R}$ and any reply resulting from the converted query can be converted back to a reply for $\mathbb{S}$. By wrapping the incoming side of an open simulation with a more general convention and its outgoing side with a more specialized convention, one gets another valid open simulation [Koenig and Shao 2021]:

Theorem 3.5. *Given $L_1 : A_1 \twoheadrightarrow B_1$ and $L_2 : A_2 \twoheadrightarrow B_2$, if $\mathbb{R}_A' \sqsubseteq \mathbb{R}_A : A_1 \Leftrightarrow A_2$, $\mathbb{R}_B \sqsubseteq \mathbb{R}_B' : B_1 \Leftrightarrow B_2$ and $L_1 \leqslant_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$, then $L_1 \leqslant_{\mathbb{R}_A' \twoheadrightarrow \mathbb{R}_B'} L_2$.*

Now, we would like to prove the "real" vertical composition generating direct refinements (simulations). Given any $L_1 \leqslant_{\mathbb{R}_{12} \twoheadrightarrow \mathbb{R}_{12}} L_2$ and $L_2 \leqslant_{\mathbb{R}_{23} \twoheadrightarrow \mathbb{R}_{23}} L_3$, if we can show the existence of simulation conventions $\mathbb{R}_{13}$ directly relating source and target semantics s.t. $\mathbb{R}_{13} \equiv \mathbb{R}_{12} \cdot \mathbb{R}_{23}$, then $L_1 \leqslant_{\mathbb{R}_{13} \twoheadrightarrow \mathbb{R}_{13}} L_3$ holds by Theorem 3.4 and Theorem 3.5, which is the desired direct refinement. This composition is illustrated in Fig. 10 where the parts enclosed by dashed boxes represent the concatenation of $L_1 \leqslant_{\mathbb{R}_{12} \twoheadrightarrow \mathbb{R}_{12}} L_2$ and $L_2 \leqslant_{\mathbb{R}_{23} \twoheadrightarrow \mathbb{R}_{23}} L_3$. The direct queries and replies are split and merged for interaction with parallelly running simulations underlying the direct refinement.

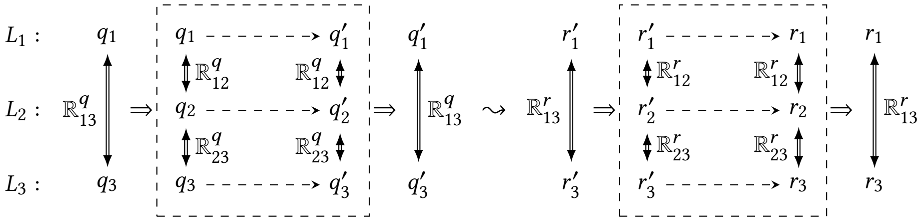
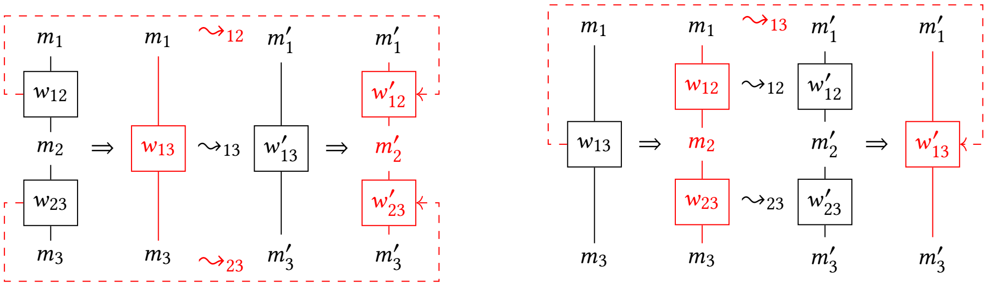


Fig. 10. Vertical Composition of Open Simulations by Refinement of Simulation Conventions



(a) $K_{13} \sqsubseteq K_{12} \cdot K_{23}$

(b) $K_{12} \cdot K_{23} \sqsubseteq K_{13}$

Fig. 11. Composition of KMRs

Since simulation conventions are parameterized by KMRs, a major obstacle to the real vertical composition of open simulations is to prove KMRs for individual simulations can be composed into a single KMR. For this, one needs to define refinements between KMRs. Given any KMRs K and L, $K \sqsubseteq L$ (i.e., K is refined by L) holds if the following is true:

$$\begin{aligned}\forall\, w_L\ m_1\ m_2,\ (m_1, m_2) \in R_L(w_L) \Rightarrow \exists\, w_K,\ (m_1, m_2) \in R_K(w_K) \wedge f_L(w_L) \subseteq f_K(w_K) \wedge \\ \forall\, w'_K\ m'_1\ m'_2,\ w_K \rightsquigarrow_K w'_K \Rightarrow (m'_1, m'_2) \in R_K(w'_K) \Rightarrow \\ \exists\, w'_L,\ w_L \rightsquigarrow_L w'_L \wedge (m'_1, m'_2) \in R_L(w'_L) \wedge f_K(w'_K) \subseteq f_L(w'_L).\end{aligned}$$

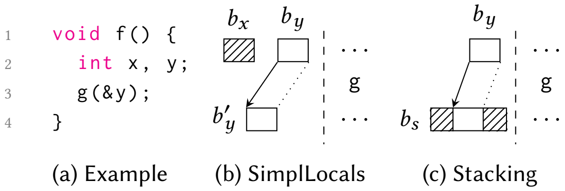
We write $K \equiv L$ to denote that K and L are equivalent, i.e., $K \sqsubseteq L$ and $L \sqsubseteq K$.

Continue with the proof of real vertical composition, i.e., proving $\mathbb{R}_{13} \equiv \mathbb{R}_{12} \cdot \mathbb{R}_{23}$. Assume $\mathbb{R}_i$ is parameterized by KMR K_i, showing the existence of $\mathbb{R}_{13}$ s.t. $\mathbb{R}_{13} \sqsubseteq \mathbb{R}_{12} \cdot \mathbb{R}_{23}$ amounts to proving a parallel refinement over the parameterizing KMRs, i.e., there exists K_{13} s.t. $K_{13} \sqsubseteq K_{12} \cdot K_{23}$ where $K_{12} \cdot K_{23} = \langle W_{12} \times W_{23}, f_{12} \times f_{23}, \rightsquigarrow_{12} \times \rightsquigarrow_{23}, R_{12} \times R_{23}\rangle$. A more intuitive interpretation is depicted in Fig. 11a where black symbols are $\forall$-quantified (assumptions we know) and red ones are $\exists$-quantified (conclusions we need to construct). Note that Fig. 11a exactly mirrors the refinement on the outgoing side in Fig. 10. For simplicity, we use w_i not only to represent worlds, but also to denote $R_i(w_i)$ (where R_i is the Kripke relation given by KMR K_i) when it connects memory states through vertical lines. A dual property we need to prove for the incoming side is shown in Fig. 11b.

In both cases in Fig. 11, we need to construct interpolating states for relating source and target memory (i.e., m'_2 in Fig. 11a and m_2 in Fig. 11b). The construction of m'_2 is especially challenging, for which we need to decompose the evolved world w'_{13} into w'_{12} and w'_{23} s.t. they are accessible from the original worlds w_{12} and w_{23}. It is not clear at all how this construction is possible because *1)* m'_2 may have many forms since Kripke *relations* are in general non-deterministic and *2)* KMRs (e.g., `injp`) introduce memory protection for external calls which may not hold after the (de-)composition.

$$\begin{tikzcd}
(b_1, o_1) \arrow[r, "\mathtt{read}"] \arrow[d, "j"'] & (b_2, o_2) \arrow[r, "\mathtt{read}"] \arrow[d, "j"'] & \cdots \\
(b'_1, o'_1) \arrow[r, "\mathtt{read}"] & (b'_2, o'_2) \arrow[r, "\mathtt{read}"] & \cdots
\end{tikzcd}$$

Fig. 12. Closure of Public Memory



```
void f() {
  int x, y;
  g(&y);
}
```

(a) Example (b) SimplLocals (c) Stacking

Fig. 13. Protection of Private Memory by `injp`

Because of the above difficulties, existing approaches either make substantial changes to semantics for constructing interpolating states, thereby destroying adequacy [Stewart et al. 2015], or do not even try to merge Kripke memory relations, but instead leave them as separate entities [Koenig and Shao 2021; Song et al. 2020]. As a result, direct refinements cannot be achieved.

4 A UNIFORM AND TRANSITIVE KRIPKE MEMORY RELATION

To overcome the challenge for vertically composing open simulations, we exploit the observation that `injp` in fact can be viewed as a most general KMR. Then, the compositionality of KMRs discussed in §3.2 is reduced to transitivity of `injp`, i.e., $\mathtt{injp} \equiv \mathtt{injp} \cdot \mathtt{injp}$.

4.1 Uniformity of `injp`

We show that `injp` is both a reasonable guarantee condition and a reasonable rely condition for all the compiler passes in CompCert. It is based on the observation that a notion of private and public memory can be derived from injections and coincides with the protection provided by `injp`.

4.1.1 Public and Private Memory via Memory Injections.

Definition 4.1. Given $m_1 \hookrightarrow^j_m m_2$, the public memory regions in m_1 and m_2 are defined as follows:

$$\begin{aligned}
&\mathtt{pub\text{-}src\text{-}mem}(j) &&= \{(b, o) \mid j(b) \neq \emptyset\};\\
&\mathtt{pub\text{-}tgt\text{-}mem}(j, m_1) &&= \{(b, o) \mid \exists b'\ o', j(b') = \lfloor (b, o') \rfloor \wedge (b', o - o') \in \mathtt{perm}(m_1, \mathtt{NA})\}.
\end{aligned}$$

By definition, a cell (b, o) is public in the source memory if it is in the domain of j, and (b, o) is public in the target memory if it is mapped by j from some valid public source memory. Any memory not public with respect to j is private. We can see that private memory corresponds exactly to unmapped and out-of-reach memory defined by `injp`, i.e., for any b and o, $(b, o) \in \mathtt{pub\text{-}src\text{-}mem}(j) \Leftrightarrow (b, o) \notin \mathtt{unmapped}(j)$ and $(b, o) \in \mathtt{pub\text{-}tgt\text{-}mem}(j, m) \Leftrightarrow (b, o) \notin \mathtt{out\text{-}of\text{-}reach}(j, m)$.

With Definition 4.1 and the properties of memory injection (see §3.1.1), we can easily prove access of pointers in a readable and public source location gets back another public location.

Lemma 4.2. *Given* $m_1 \hookrightarrow^j_m m_2$*,*

$$\begin{aligned}
&\forall\, b_1\ o_1,\ (b_1, o_1) \in \mathtt{pub\text{-}src\text{-}mem}(j) \Rightarrow (b_1, o_1) \in \mathtt{perm}(m_1, \mathtt{Readable}) \Rightarrow\\
&\qquad m_1[b_1, o_1] = \mathtt{Vptr}(b'_1, o'_1) \Rightarrow (b'_1, o'_1) \in \mathtt{pub\text{-}src\text{-}mem}(j).
\end{aligned}$$

It implies that readable public memory regions form a "closure" such that the sequences of reads are bounded inside these regions, as shown in Fig. 12. The horizontal arrows indicates a pointer value (b_{i+1}, o_{i+1}) is read from (b_i, o_i) with possible adjustment with pointer arithmetic. Note that all memory cells at (b_i, o_i)s and (b'_i, o'_i)s have `Readable` permission. By Lemma 4.2, (b_i, o_i)s are all in public regions. By Definition 4.1, the mirroring reads (b'_i, o'_i)s are also in public regions.

4.1.2 `injp` *as a Uniform Rely Condition.* `injp` is adequate for preventing external calls from interfering with internal execution for all the compiler passes of CompCert. [2] To illustrate this point, we discuss the effect of `injp` on two of CompCert's passes using Fig. 13a as an example where `g` is an external function. The first pass is `SimplLocals` which converts local variables whose memory addresses are not taken into temporary ones. As shown in Fig. 13b, `x` is turned into a temporary variable at the target level which is not visible to `g`. Therefore, `x` at the source level becomes *private data* as its block b_x is unmapped by j, thereby protected by `injp` and cannot be modified by `g`. The second pass is `Stacking` which expands the stack frames with private regions for return addresses, spilled registers, arguments, etc. Continuing with our example, the only public stack data in Fig. 13c is y. All the private data is out-of-reach, thereby protected by `injp`.

4.1.3 `injp` *as a Uniform Guarantee Condition.* For `injp` to serve as a uniform guarantee condition, it suffices to show the private memory of the environment is protected between initial calls and final replies. During an open forward simulation, all incoming values and memories are related by some initial injection j (e.g., $\vec{v_1} \hookrightarrow_v^j \vec{v_2}$ and $m_1 \hookrightarrow_m^j m_2$). In particular, the pointers in them are related by j. Therefore, any sequence of reads starting from pointers stored in the initial queries only inspect public memories in the source and target, as already shown in Fig. 12. The private (i.e., unmapped or out-of-reach) regions of the initial memories are *not modified* by internal execution. Moreover, because injection functions only grow bigger during execution but never change in value and the outgoing calls have `injp` as a rely-condition, the initially unmapped (out-of-reach) regions will stay unmapped (out-of-reach) and be protected during external calls. As a result, we conclude that `injp` is a reasonable guarantee condition for any open simulation.

4.2 Transitivity of `injp`

The goal is to show the two refinements in Fig. 11 hold when K_{ij} = `injp`, i.e., `injp` ≡ `injp`·`injp`. As discussed in §3.2, the critical step is to construct interpolating memory states that transitively relate source and target states. The construction is based on two observations: *1)* the memory injections deterministically decide the value and permissions of public memory because they encode *partial functional transformations* on memory states, and *2)* any memory not in the domain or range of the partial functions is protected (private) and unchanged throughout external calls. Although the proof is quite involved, the result can be reused for all compiler passes thanks to `injp`'s uniformity.

$$\begin{tikzcd}
m_1 \arrow[rr] \arrow[d, "j_{12}"'] & & m_1' \arrow[dd, "j_{13}'"] & & m_1 \arrow[rr] \arrow[d, "j_{12}"'] & & m_1' \arrow[d, "\rightsquigarrow_{\texttt{injp}}\ j_{12}'"'] \arrow[dd, bend left, "j_{13}'"] \\
m_2 \arrow[d, "j_{23}"'] & \rightsquigarrow_{\texttt{injp}} & & \Rightarrow & m_2 \arrow[rr] \arrow[d, "j_{23}"'] & & m_2' \arrow[d, "\rightsquigarrow_{\texttt{injp}}\ j_{23}'"'] \\
m_3 \arrow[rr] & & m_3' & & m_3 \arrow[rr] & & m_3'
\end{tikzcd}$$

Fig. 14. Construction of Interpolating States

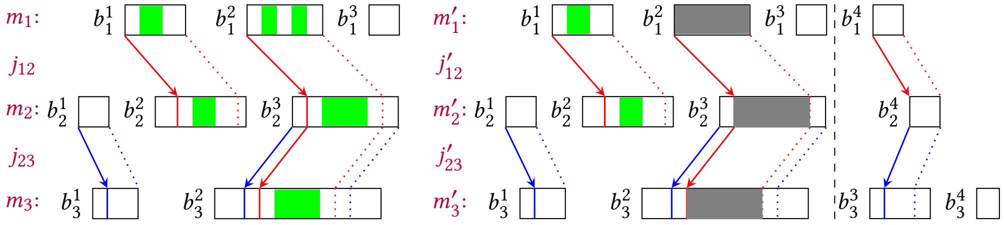
4.2.1 `injp` ⊑ `injp` • `injp`*.* By definition, we need to prove the following lemma:

Lemma 4.3. `injp` ⊑ `injp` • `injp` *holds. That is,*

$$\forall j_{12}\ j_{23}\ m_1\ m_2\ m_3,\ m_1 \hookrightarrow_m^{j_{12}} m_2 \Rightarrow m_2 \hookrightarrow_m^{j_{23}} m_3 \Rightarrow \exists j_{13},\ m_1 \hookrightarrow_m^{j_{13}} m_3\ \land$$
$$\forall m_1'\ m_3'\ j_{13}',\ (j_{13}, m_1, m_3) \rightsquigarrow_{\texttt{injp}} (j_{13}', m_1', m_3') \Rightarrow m_1' \hookrightarrow_m^{j_{13}'} m_3' \Rightarrow$$
$$\exists m_2'\ j_{12}'\ j_{23}', (j_{12}, m_1, m_2) \rightsquigarrow_{\texttt{injp}} (j_{12}', m_1', m_2') \land m_1' \hookrightarrow_m^{j_{12}'} m_2'$$
$$\land (j_{23}, m_2, m_3) \rightsquigarrow_{\texttt{injp}} (j_{23}', m_2', m_3') \land m_2' \hookrightarrow_m^{j_{23}'} m_3'.$$

This lemma conforms to the graphic representation in Fig. 11a. To prove it, an obvious choice is to pick $j_{13} = j_{23} \cdot j_{12}$. Then, we are left to prove the existence of interpolating state m_2' and the

[2] In fact, the properties in Definition 3.3 are exactly from CompCert's assumptions on external calls.



(a) At the External Call

(b) After the External Call

Fig. 15. Constructing of an Interpolating Memory State

memory and accessibility relations as shown in Fig. 14. By definition, m'_2 consists of memory blocks newly allocated with respect to m_2 and blocks that already exist in m_2. The latter can be further divided into public and private memory regions with respect to injections j_{12} and j_{23}. Then, m'_2 is constructed following the ideas that *1)* the public and newly allocated memory should be projected from the updated source memory m'_1 by j'_{12}, and *2)* the private memory is protected by `injp` and should be copied over from m_2 to m'_2.

We use the concrete example in Fig. 15 to motivate the construction of m'_2. Here, the white and green areas correspond to locations in `perm(_, NA)` (with at least some permission) and in `perm(_, Readable)` (with at least readable permission), respectively. Given $m_1 \hookrightarrow_m^{j_{12}} m_2$, $m_2 \hookrightarrow_m^{j_{23}} m_3$ and $(j_{23} \cdot j_{12}, m_1, m_3) \leadsto_{\mathtt{injp}} (j'_{13}, m'_1, m'_3)$, we need to define j'_{12} and j'_{23} and build m'_2 satisfying $m'_1 \hookrightarrow_m^{j'_{12}} m'_2$, $m'_2 \hookrightarrow_m^{j'_{23}} m'_3$, $(j_{12}, m_1, m_2) \leadsto_{\mathtt{injp}} (j'_{12}, m'_1, m'_2)$, and $(j_{23}, m_2, m_3) \leadsto_{\mathtt{injp}} (j'_{23}, m'_2, m'_3)$. m'_1 and m'_3 are expansions of m_1 and m_3 with new blocks and possible modification to the public regions of m_1 and m_3. Here, m'_1 has a new block b_1^4 and m'_3 has two new block b_3^3 and b_3^4.

We first fix j'_{12}, j'_{23} and the shape of blocks in m'_2. We begin with m_2 and introduce a newly allocated block b_2^4 whose shape matches b_1^4 in m'_1. Then, j'_{12} is obtained by expanding j_{12} with identity mapping from b_1^4 to b_2^4. Furthermore, j'_{23} is also expanded with a mapping from b_2^4 to a block in m'_3; this mapping is determined by j'_{13}.

We then set the values and permissions for memory cells in m'_2 so that it satisfies injection and the `unchanged-on` properties for readable memory regions implied by $(j_{12}, m_1, m_2) \leadsto_{\mathtt{injp}} (j'_{12}, m'_1, m'_2)$ and $(j_{23}, m_2, m_3) \leadsto_{\mathtt{injp}} (j'_{23}, m'_2, m'_3)$. The values and permissions for newly allocated blocks are obviously mapped from m'_1 by j'_{12}. Those for old blocks are fixed as follows. By memory protection provided in $(j_{23} \cdot j_{12}, m_1, m_3) \leadsto_{\mathtt{injp}} (j'_{13}, m'_1, m'_3)$, the only memory cells in m_1 that may have been modified in m'_1 are those mapped all the way to m_3 by $j_{23} \cdot j_{12}$, while the cells in m_3 that may be modified in m'_3 must be in the image of $j_{23} \cdot j_{12}$. To match this fact, the only old memory regions in m'_2 whose values and permissions may be modified are those both in the image of j_{12} and the domain of j_{23}. Those are the public memory with respect to j_{12} and j_{23} and displayed as the gray areas in Fig. 15b. Following idea *1)* above, the values and permissions in those regions are projected from m'_1 by applying the injection function j_{12}. Note that there is an exception: values in read-only public regions are copied over from m_2. Following idea *2)* above, the remaining old memory regions are private with respect to j_{12} and j_{23} and should have the same values and permissions as in m_2.

Note that the accessibility relations $(j_{12}, m_1, m_2) \leadsto_{\mathtt{injp}} (j'_{12}, m'_1, m'_2)$ and $(j_{23}, m_2, m_3) \leadsto_{\mathtt{injp}} (j'_{23}, m'_2, m'_3)$ can be derived from $(j_{23} \cdot j_{12}, m_1, m_3) \leadsto_{\mathtt{injp}} (j'_{13}, m'_1, m'_3)$ because the latter enforces *stronger* protection than the former. This is due to unmapped and out-of-reach regions getting *bigger* as memory injections get composed. For example, in Fig. 15, b_1^1 is mapped by j_{12} but becomes unmapped by $j_{23} \cdot j_{12}$; the image of b_2^1 in b_3^1 is in reach by j_{23} but becomes out-of-reach by $j_{23} \cdot j_{12}$.

Table 1. Significant Passes of CompCert

Languages/Passes	Outgoing $\twoheadrightarrow$ Incoming	Language/Pass	Outgoing $\twoheadrightarrow$ Incoming
Clight	$\mathcal{C} \twoheadrightarrow \mathcal{C}$	Constprop	$\texttt{ro}\cdot\texttt{c}_{\texttt{injp}} \twoheadrightarrow \texttt{ro}\cdot\texttt{c}_{\texttt{injp}}$
Self-Sim	$\texttt{ro}\cdot\texttt{c}_{\texttt{injp}} \twoheadrightarrow \texttt{ro}\cdot\texttt{c}_{\texttt{injp}}$	CSE	$\texttt{ro}\cdot\texttt{c}_{\texttt{injp}} \twoheadrightarrow \texttt{ro}\cdot\texttt{c}_{\texttt{injp}}$
SimplLocals	$\texttt{c}_{\texttt{injp}} \twoheadrightarrow \texttt{c}_{\texttt{inj}}$	Deadcode	$\texttt{ro}\cdot\texttt{c}_{\texttt{injp}} \twoheadrightarrow \texttt{ro}\cdot\texttt{c}_{\texttt{injp}}$
Csharpminor	$\mathcal{C} \twoheadrightarrow \mathcal{C}$	Unusedglob	$\texttt{c}_{\texttt{inj}} \twoheadrightarrow \texttt{c}_{\texttt{inj}}$
Cminorgen	$\texttt{c}_{\texttt{injp}} \twoheadrightarrow \texttt{c}_{\texttt{inj}}$	Allocation	$\texttt{wt}\cdot\texttt{c}_{\texttt{ext}}\cdot\texttt{CL} \twoheadrightarrow \texttt{wt}\cdot\texttt{c}_{\texttt{ext}}\cdot\texttt{CL}$
Cminor	$\mathcal{C} \twoheadrightarrow \mathcal{C}$	**LTL**	$\mathcal{L} \twoheadrightarrow \mathcal{L}$
Selection	$\texttt{wt}\cdot\texttt{c}_{\texttt{ext}} \twoheadrightarrow \texttt{wt}\cdot\texttt{c}_{\texttt{ext}}$	Tunneling	$\texttt{ltl}_{\texttt{ext}} \twoheadrightarrow \texttt{ltl}_{\texttt{ext}}$
CminorSel	$\mathcal{C} \twoheadrightarrow \mathcal{C}$	**Linear**	$\mathcal{L} \twoheadrightarrow \mathcal{L}$
RTLgen	$\texttt{c}_{\texttt{ext}} \twoheadrightarrow \texttt{c}_{\texttt{ext}}$	Stacking	$\texttt{ltl}_{\texttt{injp}}\cdot\texttt{LM} \twoheadrightarrow \texttt{LM}\cdot\texttt{mach}_{\texttt{inj}}$
RTL	$\mathcal{C} \twoheadrightarrow \mathcal{C}$	**Mach**	$\mathcal{M} \twoheadrightarrow \mathcal{M}$
Self-Sim	$\texttt{c}_{\texttt{inj}} \twoheadrightarrow \texttt{c}_{\texttt{inj}}$	Asmgen	$\texttt{mach}_{\texttt{ext}}\cdot\texttt{MA} \twoheadrightarrow \texttt{mach}_{\texttt{ext}}\cdot\texttt{MA}$
Tailcall	$\texttt{c}_{\texttt{ext}} \twoheadrightarrow \texttt{c}_{\texttt{ext}}$	**Asm**	$\mathcal{A} \twoheadrightarrow \mathcal{A}$
Inlining	$\texttt{c}_{\texttt{injp}} \twoheadrightarrow \texttt{c}_{\texttt{inj}}$	Self-Sim	$\texttt{asm}_{\texttt{inj}} \twoheadrightarrow \texttt{asm}_{\texttt{inj}}$
Self-Sim	$\texttt{c}_{\texttt{injp}} \twoheadrightarrow \texttt{c}_{\texttt{injp}}$	Self-Sim	$\texttt{asm}_{\texttt{injp}} \twoheadrightarrow \texttt{asm}_{\texttt{injp}}$

4.2.2 $\texttt{injp}\cdot\texttt{injp} \sqsubseteq \texttt{injp}$.

By definition, we need to prove:

LEMMA 4.4. $\texttt{injp}\cdot\texttt{injp} \sqsubseteq \texttt{injp}$ *holds. That is,*

$$\begin{aligned}
&\forall j_{13}\ m_1\ m_3,\ m_1 \hookrightarrow_m^{j_{13}} m_3 \Rightarrow \exists j_{12}\ j_{23}\ m_2,\ m_1 \hookrightarrow_m^{j_{12}} m_2 \land m_2 \hookrightarrow_m^{j_{23}} m_3 \land \\
&\quad \forall m_1'\ m_2'\ m_3'\ j_{12}'\ j_{23}',\ (j_{12}, m_1, m_2) \leadsto_{\texttt{injp}} (j_{12}', m_1', m_2') \Rightarrow (j_{23}, m_2, m_3) \leadsto_{\texttt{injp}} (j_{23}', m_2', m_3') \Rightarrow \\
&\qquad m_1' \hookrightarrow_m^{j_{12}'} m_2' \Rightarrow m_2' \hookrightarrow_m^{j_{23}'} m_3' \Rightarrow \exists j_{13}',\ (j_{13}, m_1, m_3) \leadsto_{\texttt{injp}} (j_{13}', m_1', m_3') \land m_1' \hookrightarrow_m^{j_{13}'} m_3'.
\end{aligned}$$

This lemma conforms to Fig. 11b. To prove it, we pick j_{12} to be an partial identity injection $(j_{12}(b) = \lfloor b, 0 \rfloor$ when $j_{13}(b) \neq \emptyset)$, $j_{23} = j_{13}$ and $m_2 = m_1$. Then the lemma is reduced to proving the existence of j_{13}' that satisfies $(j_{13}, m_1, m_3) \leadsto_{\texttt{injp}} (j_{13}', m_1', m_3')$ and $m_1' \hookrightarrow_m^{j_{13}'} m_3'$. By picking $j_{13}' = j_{12}' \cdot j_{23}'$, we can easily prove these properties by exploiting the properties of $\texttt{injp}$.

5 DERIVATION OF THE DIRECT REFINEMENT FOR COMPCERT

In this section, we discuss the proofs and composition of open simulations for the compiler passes of CompCert into the direct refinement $\leqslant_{\texttt{ac}}$ following the ideas discussed in §2.2. CompCert compiles Clight programs into Asm programs through 19 passes [Leroy 2023], including several optimization passes working on the RTL intermediate language. First, we prove the open simulations for all these passes with appropriate simulation conventions. In particular, we directly reuse the proofs of non-optimizing passes in CompCertO and update the proofs of optimizing passes with semantic invariants. Second, we prove a collection of properties for refining simulation conventions in preparation for vertical composition. Those properties enable absorption of KMRs into injp and composition of semantic invariants. They rely critically on transitivity of injp. Finally, we vertically compose the simulations and refine the incoming and outgoing simulation conventions into a single simulation convention $\mathbb{C}$, thereby establishing $\leqslant_{\mathbb{C}}$ as the top-level refinement $\leqslant_{\texttt{ac}}$.

5.1 Open Simulation of Individual Passes

We list the compiler passes and their simulation types in Table 1 (passes on the right follow the passes on the left) together with their source and target languages and interfaces (in bold fonts). The passes in black are reused from CompCertO, while those in red are reproved optimizing passes. The passes in blue are *self-simulating* passes we inserted; they will be used in §5.3 for refining composed simulation conventions. Note that we have omitted passes with the identity simulation convention (i.e., simulations of the form $L_1 \leqslant_{\mathtt{id}} L_2$) in Table 1 as they do not affect the proofs. [3]

5.1.1 Simulation Conventions and Semantic Invariants. We first introduce relevant simulation conventions and semantic invariants shown in Table 1. The simulation conventions $\mathtt{c}_K : \mathcal{C} \Leftrightarrow \mathcal{C}$, $\mathtt{ltl}_K : \mathcal{L} \Leftrightarrow \mathcal{L}$, $\mathtt{mach}_K : \mathcal{M} \Leftrightarrow \mathcal{M}$, and $\mathtt{asm}_K : \mathcal{A} \Leftrightarrow \mathcal{A}$ relate the same language interfaces with queries and replies native to the associated intermediate languages. They are parameterized by a KMR K to allow different compiler passes to have different assumptions on memory evolution. Conceptually, this parameterization is unnecessary as we can simply use `injp` for every pass due to its uniformity (as discussed in §4.1). Nevertheless, it is useful because the compiler proofs become simpler and more natural with the least restrictive KMRs which may be weaker than `injp`. CompCertO defines several KMRs weaker than `injp`: `id` is used when memory is unchanged; `ext` is used when the source and target memory share the same structure; `inj` is a simplified version of `injp` without its memory protection. The simulation conventions $\mathtt{CL} : \mathcal{C} \Leftrightarrow \mathcal{L}$, $\mathtt{LM} : \mathcal{L} \Leftrightarrow \mathcal{M}$ and $\mathtt{MA} : \mathcal{M} \Leftrightarrow \mathcal{A}$ capture the calling convention of CompCert: `CL` relates C-level queries and replies to those in the `LTL` language where the arguments are distributed to abstract stack slots; `LM` further relates abstract stack slots with states on an architecture independent machine; `MA` relates this state to registers and memory in the assembly language (X86 assembly in our case). As discussed before, some refinements rely on invariants on the source semantics. The semantic invariant `wt` enforces that arguments and return values of function calls respect function signatures. `ro` is critical for ensuring the correctness of optimizations, which will be discussed next.

5.1.2 Open Simulation of Optimizations. The optimizing passes `Constprop`, `CSE` and `Deadcode` perform constant propagation, common subexpression elimination and dead code elimination, respectively. They make use of a static value analysis algorithm for collecting information of variables during the execution. For each function, this algorithm starts with the known initial values of read-only (constant) global variables. It simulates the function execution to analyze the values of global or local variables after executing each instruction. In particular, for global *constant* variables, their references at any point should have the initial values of constants. For local variables stored on the stack, their references may have initial values or may not if interfered by other function calls. When the analysis encounters a call to another function, it checks whether the address of current stack frame is leaked to the callee directly through arguments or indirectly through pointers in memory. If not, then the stack frame is considered *unreachable* from its callee. Consequently, the references to local variables on unreachable stack frames after function calls remain to be their initial values. Based on this analysis, the three passes then identify and perform optimizations.

```
const int key = 42;
void foo(int*);
int double_key() {
  int a = key;
  foo(&key);
  return a + key;
}
```

(a) Source Program

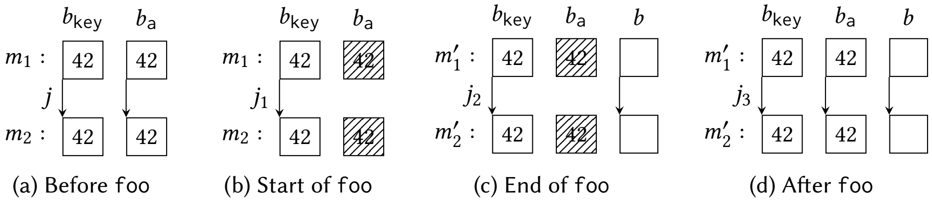
```
const int key = 42;
void foo(int*);
int double_key() {
  int a = 42;
  foo(&key);
  return 84;
}
```

(b) Target Program

Fig. 16. An Example of Constant Propagation

Most of the proofs of closed simulations for those passes can be adapted to open simulation straightforwardly. The only and main difficulty is to prove that information derived from static analysis is consistent with the dynamic memory states in incoming queries and after external calls return. We introduce the semantic invariant `ro` and combine it with `injp` to ensure this consistency. The above optimization passes all use $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}}$ as their simulation conventions (because `RTL`

[3] The omitted passes are `Cshmgen`, `Renumber`, `Linearize`, `CleanupLabels` and `Debugvar`.



(a) Before foo (b) Start of foo (c) End of foo (d) After foo

Fig. 17. Memory Injections from Call to Return of foo

conforms to the C interface). The adaptation of optimization proofs for those passes is similar. As an example, we only discuss constant propagation whose correctness theorem is stated as follows:

Lemma 5.1. $\forall (M\ M' : \mathsf{RTL}), \mathsf{Constprop}(M) = M' \Rightarrow [\![M]\!] \leqslant_{\mathsf{ro}\cdot\mathsf{c}_{\mathsf{injp}}} [\![M']\!]$.

Instead of presenting its proof, we illustrate how ro and injp help establish the open simulation for Constprop through a concrete example as depicted in Fig. 16. This example covers optimization for both global constants (e.g., key) and local variables (e.g., a). By static analysis of Fig. 16a, *1)* key contains 42 at line 4 because key is a constant global variable and, *2)* both key and a contain 42 after the external call to foo returns to line 6. Here, the analysis confirms key has the value 42 because foo (if well-behaved) will not modify a constant global variable. Furthermore, a has the value 42 because it resides in the stack frame of double_key which is unreachable from foo (in fact, a is the only variable in the frame). As a result, the source program is optimized into Fig. 16b.

We first show that ro guarantees the dynamic values of global constants are consistent with static analysis. That is, global variables are correct in incoming memory and are protected during external calls. ro is defined as follows:

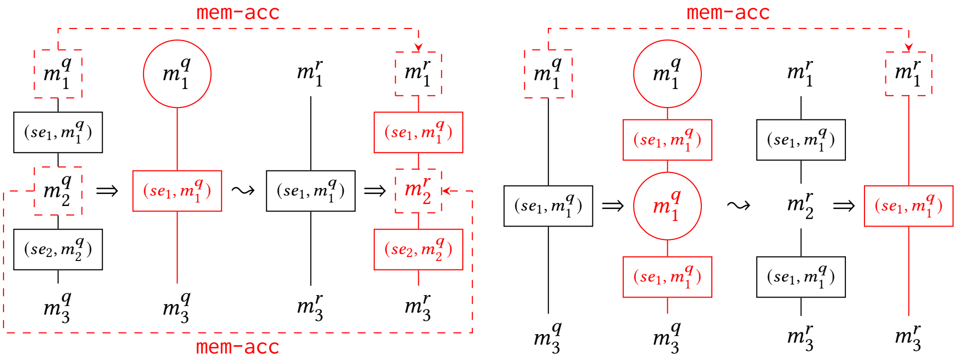
Definition 5.2. $\mathsf{ro} : C \Leftrightarrow C = \langle W_{\mathsf{ro}}, \mathbb{R}^q_{\mathsf{ro}}, \mathbb{R}^r_{\mathsf{ro}} \rangle$ where $W_{\mathsf{ro}} = (\mathsf{symtbl} \times \mathsf{mem})$ and

$$\mathbb{R}^q_{\mathsf{ro}}(se, m) = \{(v_f[sg](\vec{v})@m, v_f[sg](\vec{v})@m) \mid \mathsf{ro\text{-}valid}(se, m)\}$$
$$\mathbb{R}^r_{\mathsf{ro}}(se, m) = \{(res@m', res@m') \mid \mathsf{mem\text{-}acc}(m, m')\}$$

Note that although ro takes the form of a simulation convention, it only relates the same queries and replies, i.e., it only enforces invariants on the source side. This kind of simulation conventions are what we called *semantic invariants*. A symbol table *se* (of type symtbl) is provided together with memory, so that the semantics can locate the memory blocks and initial values of global definitions. $\mathsf{ro\text{-}valid}(se, m)$ states that the values of global constant variables in the incoming memory m are the same as their initial values. Therefore, the optimization of key into 42 at line 4 of Fig. 16a is correct. For the call to foo, $\mathsf{mem\text{-}acc}(m, m')$ ensures that read-only values in memory are unchanged, therefore ro-valid is preserved across external calls (i.e., $\mathsf{ro\text{-}valid}(se, m) \Rightarrow \mathsf{mem\text{-}acc}(m, m') \Rightarrow \mathsf{ro\text{-}valid}(se, m')$). As a result, replacing key with 42 at line 6 makes sense.

We then show that injp guarantees the dynamic values of unreachable local variables are consistent with static analysis. That is, unreachable stack values are unchanged by external calls. This protection is realized by injp with *shrinking* memory injections. Fig. 17 shows the protection of a when calling foo. Before the external call to foo, the source blocks b_{a} and b_{key} are *mapped* to target blocks by the current injection j. The analysis determines that the argument and memory passed to foo do not contain any pointer to b_{a}. Therefore, we can simply remove b_{a} from j to get a shrunk yet valid memory injection j_1. Then, b_{a} is protected during the call to foo. b_{a} is added back to the injection after foo returns and the simulation continues.

Finally, Unusedglob which removes unused static global variables is verified by assuming that global symbols remain the same throughout the compilation and with a weaker KMR inj.



(a) $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \sqsubseteq \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}}$

(b) $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \sqsubseteq \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}}$

Fig. 18. Transitivity of $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}}$

5.2 Properties for Refining Simulation Conventions

We present properties necessary for refining the composed simulation conventions in Table 1.

5.2.1 *Commutativity of KMRs and Structural Conventions.*

Lemma 5.3. *For* $\mathtt{Z} \in \{\mathtt{CL}, \mathtt{LM}, \mathtt{MA}\}$ *and* $K \in \{\mathtt{ext}, \mathtt{inj}, \mathtt{injp}\}$ *we have* $\mathtt{X}_K \cdot \mathtt{Z} \sqsubseteq \mathtt{Z} \cdot \mathtt{Y}_K$.

This lemma is provided by CompCertO [Koenig and Shao 2021]. $\mathtt{X}$ and $\mathtt{Y}$ denote the simulation conventions for the source and target languages of $\mathtt{Z}$, respectively (e.g., $\mathtt{X} = \mathtt{c}$ and $\mathtt{Y} = \mathtt{ltl}$ when $\mathtt{Z} = \mathtt{CL}$). If $K = \mathtt{injp}$ we get $\mathtt{c}_{\mathtt{injp}} \cdot \mathtt{CL} \sqsubseteq \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{injp}}$. This lemma indicates at the outgoing (incoming) side a convention lower (higher) than $\mathtt{CL}$, $\mathtt{LM}$ and $\mathtt{MA}$ may be lifted over them to a higher position (pushed down to a lower position).

5.2.2 *Absorption of KMRs into* `injp`*.*

The lemma below is needed for absorbing KMRs into `injp`:

Lemma 5.4. *For any* $\mathbb{R}$, $(1)\mathbb{R}_{\mathtt{injp}} \cdot \mathbb{R}_{\mathtt{injp}} \equiv \mathbb{R}_{\mathtt{injp}}$ $(2)\mathbb{R}_{\mathtt{injp}} \sqsubseteq \mathbb{R}_{\mathtt{inj}}$ $(3)\mathbb{R}_{\mathtt{injp}} \cdot \mathbb{R}_{\mathtt{inj}} \cdot \mathbb{R}_{\mathtt{injp}} \sqsubseteq \mathbb{R}_{\mathtt{injp}}$ $(4)\mathbb{R}_{\mathtt{inj}} \cdot \mathbb{R}_{\mathtt{inj}} \sqsubseteq \mathbb{R}_{\mathtt{inj}}$ $(5)\mathbb{R}_{\mathtt{ext}} \cdot \mathbb{R}_{\mathtt{inj}} \equiv \mathbb{R}_{\mathtt{inj}}$ $(6)\mathbb{R}_{\mathtt{inj}} \cdot \mathbb{R}_{\mathtt{ext}} \equiv \mathbb{R}_{\mathtt{inj}}$ $(7)\mathbb{R}_{\mathtt{ext}} \cdot \mathbb{R}_{\mathtt{ext}} \equiv \mathbb{R}_{\mathtt{ext}}$.

The simulation convention $\mathbb{R}$ is parameterized over a KMR. Property (1) is a direct consequence of $\mathtt{injp} \cdot \mathtt{injp} \equiv \mathtt{injp}$, which is critical for merging simulations using `injp`. The remaining ones either depend on transitivity of `injp`, or trivially hold as shown by Koenig and Shao [2021].

5.2.3 *Composition of Semantic Invariants.*

Lastly, we also need to handle the two semantic invariants `ro` and `wt`. They cannot be absorbed into `injp` because their assumptions are fundamentally different. Therefore, our goal is to permute them to the top-level and merge any duplicated copies. The following lemmas enable elimination and permutation of `wt`:

Lemma 5.5. *For any* $\mathbb{R}_K : C \Leftrightarrow C$, *we have* $(1)\mathbb{R}_K \cdot \mathtt{wt} \equiv \mathtt{wt} \cdot \mathbb{R}_K \cdot \mathtt{wt}$ *and* $(2)\mathbb{R}_K \cdot \mathtt{wt} \equiv \mathtt{wt} \cdot \mathbb{R}_K$.

`ro` is more difficult to handle as it does not commute with arbitrary simulation conventions. To eliminate redundant `ro`, we piggyback `ro` onto `injp` and prove the following transitivity property:

Lemma 5.6. $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \equiv \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}}$

Its proof is similar to that for $\mathtt{c}_{\mathtt{injp}} \equiv \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{c}_{\mathtt{injp}}$ but with additional reasoning for establishing `ro`. A graphic presentation of the proof is given in Fig. 18 which mirrors Fig. 11. We focus on the additional reasoning and have omitted the $\rightsquigarrow_{\mathtt{injp}}$ relations and the worlds for `injp` in Fig. 18. Note that by definition the worlds (se, m) for $\mathtt{ro}$ do not evolve like those for $\mathtt{injp}$. A red circle around a memory state m indicates the necessity to prove that $\mathtt{ro\text{-}valid}$ in $\mathbb{R}^q_{\mathtt{ro}}$ holds for m. The $\mathtt{mem\text{-}acc}$ relations over dashed arrows are the properties over replies in $\mathbb{R}^r_{\mathtt{ro}}$ and must also be verified.

The above additional properties are proved based on two observations. First, the properties for queries (i.e., $\mathtt{ro\text{-}valid}$) are propagated in refinement along with copying of memory states. For example, to prove the refinement in Fig. 18a, we are given $\mathtt{ro\text{-}valid}(se_1, m_1^q)$ and $\mathtt{ro\text{-}valid}(se_2, m_2^q)$ according to the initial $\mathbb{R}^q_{\mathtt{ro}}$ relations. By choosing (se_1, m_1^q) to be the world for the composed $\mathbb{R}^q_{\mathtt{ro}}$, $\mathtt{ro\text{-}valid}(se_1, m_1^q)$ holds trivially for m_1^q in the circle. To prove the refinement in Fig. 18b, we need to prove that the interpolating memory state after the initial decomposition satisfies $\mathbb{R}^q_{\mathtt{ro}}$. By choosing m_1^q to be this state (in the middle circle in Fig. 18b and according to the proof of Lemma 4.4), $\mathtt{ro\text{-}valid}(se_1, m_1^q)$ follows directly from the initial assumption. Second, the properties for replies (i.e., $\mathtt{mem\text{-}acc}$) have already been encoded into $\leadsto_{\mathtt{injp}}$ by Definition 3.3. For example, m_2^r in Fig. 18a is constructed by following exactly Lemma 4.3. Therefore, $\mathtt{mem\text{-}acc}(m_2^q, m_2^r)$ trivially holds.

Finally, at the top level, we need $\mathtt{ro}$ and $\mathtt{wt}$ to commute which is straightforward to prove:

Lemma 5.7. $\mathtt{ro} \cdot \mathtt{wt} \equiv \mathtt{wt} \cdot \mathtt{ro}$

5.3 Proving the Direct Open Simulation for CompCert

We first insert self-simulations into the compiler passes, as shown in Table 1. This is to supply extra $\mathbb{R}_{\mathtt{inj}}$, $\mathbb{R}_{\mathtt{injp}}$, and $\mathtt{ro}$ for absorbing $\mathbb{R}_{\mathtt{ext}}$ ($\mathbb{R}_{\mathtt{inj}}$) into $\mathbb{R}_{\mathtt{inj}}$ ($\mathbb{R}_{\mathtt{injp}}$) by properties in Lemma 5.4 and for transitive composition of $\mathtt{ro}$. Self-simulations are obtained by the following lemma:

Theorem 5.8. *If p is a program written in* $\mathtt{Clight}$ *or* $\mathtt{RTL}$ *and* $\mathbb{R} \in \{\mathtt{ro}, \mathtt{c}_{\mathtt{ext}}, \mathtt{c}_{\mathtt{inj}}, \mathtt{c}_{\mathtt{injp}}\}$, *or p is written in* $\mathtt{Asm}$ *and* $\mathbb{R} \in \{\mathtt{asm}_{\mathtt{ext}}, \mathtt{asm}_{\mathtt{inj}}, \mathtt{asm}_{\mathtt{injp}}\}$, *then* $[[p]] \leqslant_{\mathbb{R} \twoheadrightarrow \mathbb{R}} [[p]]$ *holds.*

We unify the conventions at the incoming and outgoing sides. We start with the simulation $L_1 \leqslant_{\mathbb{R} \twoheadrightarrow \mathbb{S}} L_2$ which is the transitive composition of compiler passes in Table 1 where

$$\begin{aligned}\mathbb{R} = {} & \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \\ & \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{ltl}_{\mathtt{injp}} \cdot \mathtt{LM} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA} \cdot \mathtt{asm}_{\mathtt{inj}} \cdot \mathtt{asm}_{\mathtt{injp}} \\ \mathbb{S} = {} & \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \\ & \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM} \cdot \mathtt{mach}_{\mathtt{inj}} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA} \cdot \mathtt{asm}_{\mathtt{inj}} \cdot \mathtt{asm}_{\mathtt{injp}}.\end{aligned}$$

We then find two sequences of refinements $\mathbb{C} \sqsubseteq \mathbb{R}_n \sqsubseteq \ldots \sqsubseteq \mathbb{R}_1 \sqsubseteq \mathbb{R}$ and $\mathbb{S} \sqsubseteq \mathbb{S}_1 \sqsubseteq \ldots \sqsubseteq \mathbb{S}_m \sqsubseteq \mathbb{C}$, by which and Theorem 3.5 we get the simulation $L_1 \leqslant_{\mathbb{C} \twoheadrightarrow \mathbb{C}} L_2$. The direct simulation convention is $\mathbb{C} = \mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{CAinjp} \cdot \mathtt{asm}_{\mathtt{injp}}$. $\mathtt{ro}$ enables optimizations at C level while $\mathtt{wt}$ ensures well-typedness. The definition of $\mathtt{CAinjp}$ has already been discussed informally in §2.1; its formal definition is given in the technical report [Zhang et al. 2023b]. The last $\mathtt{asm}_{\mathtt{injp}}$ is irrelevant as assembly code is self-simulating by Theorem 5.8. The final correctness theorem is shown below:

Theorem 5.9. *Compilation in CompCert is correct in terms of open simulations,*

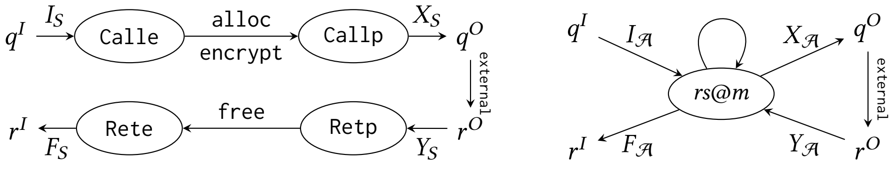
$$\forall\ (M : \mathtt{Clight})\ (M' : \mathtt{Asm}),\ \mathtt{CompCert}(M) = M' \Rightarrow [[M]] \leqslant_{\mathbb{C}} [[M']].$$

We explain how the refinements are carried out at the outgoing side. Refinements at the incoming side are similar. The following is the sequence of refined simulation conventions $\mathbb{C} \sqsubseteq \mathbb{R}_n \sqsubseteq \ldots \sqsubseteq \mathbb{R}_1 \sqsubseteq \mathbb{R}$. It begins with $\mathbb{R}$ and ends with $\mathbb{C}$.

(1) $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}}$
$\cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{ltl}_{\mathtt{injp}} \cdot \mathtt{LM} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA} \cdot \mathtt{asm}_{\mathtt{inj}} \cdot \mathtt{asm}_{\mathtt{injp}}$

(2) $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}}$
$\cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{ltl}_{\mathtt{injp}} \cdot \mathtt{LM} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA} \cdot \mathtt{asm}_{\mathtt{inj}} \cdot \mathtt{asm}_{\mathtt{injp}}$

(3) $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{injp}}$
$\cdot \mathtt{c}_{\mathtt{inj}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{ltl}_{\mathtt{injp}} \cdot \mathtt{LM} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA} \cdot \mathtt{asm}_{\mathtt{inj}} \cdot \mathtt{asm}_{\mathtt{injp}}$



(a) L_S (b) $[[\texttt{server\_opt.s}]]$

Fig. 19. Specification and Open Semantics of `server_opt.s`

(4) $\texttt{wt}\cdot\texttt{ro}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{c}_{\texttt{inj}}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{ro}\cdot\texttt{c}_{\texttt{injp}}$
$\cdot\texttt{c}_{\texttt{inj}}\cdot\texttt{c}_{\texttt{ext}}\cdot\texttt{CL}\cdot\texttt{ltl}_{\texttt{ext}}\cdot\texttt{ltl}_{\texttt{injp}}\cdot\texttt{LM}\cdot\texttt{mach}_{\texttt{ext}}\cdot\texttt{MA}\cdot\texttt{asm}_{\texttt{inj}}\cdot\texttt{asm}_{\texttt{injp}}$
(5) $\texttt{wt}\cdot\texttt{ro}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{c}_{\texttt{inj}}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{ro}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{c}_{\texttt{inj}}\cdot\texttt{c}_{\texttt{ext}}\cdot\texttt{c}_{\texttt{ext}}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{c}_{\texttt{ext}}\cdot\texttt{c}_{\texttt{inj}}\cdot\texttt{CL}\cdot\texttt{LM}\cdot\texttt{MA}\cdot\texttt{asm}_{\texttt{injp}}$
(6) $\texttt{wt}\cdot\texttt{ro}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{ro}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{CL}\cdot\texttt{LM}\cdot\texttt{MA}\cdot\texttt{asm}_{\texttt{injp}}$
(7) $\texttt{wt}\cdot\texttt{ro}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{ro}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{CL}\cdot\texttt{LM}\cdot\texttt{MA}\cdot\texttt{asm}_{\texttt{injp}}$
(8) $\texttt{wt}\cdot\texttt{ro}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{CL}\cdot\texttt{LM}\cdot\texttt{MA}\cdot\texttt{asm}_{\texttt{injp}}$
(9) $\texttt{ro}\cdot\texttt{wt}\cdot\texttt{c}_{\texttt{injp}}\cdot\texttt{CL}\cdot\texttt{LM}\cdot\texttt{MA}\cdot\texttt{asm}_{\texttt{injp}}$
(10) $\texttt{ro}\cdot\texttt{wt}\cdot\texttt{CAinjp}\cdot\texttt{asm}_{\texttt{injp}}$

In each line, the letters in red are simulation conventions transformed by the refinement operation at that step. In step (1), we merge simulation conventions and semantic invariants by property (1) and (5-7) in Lemma 5.4 and by Lemma 5.6. In steps (2-3), we move and eliminate `wt` by Lemmas 5.5 and 5.7. In step (4), we lift conventions to higher positions by Lemma 5.3. In step (5), we absorb $\texttt{c}_{\texttt{ext}}$ into $\texttt{c}_{\texttt{inj}}$ and turns $\texttt{c}_{\texttt{inj}}$ into $\texttt{c}_{\texttt{injp}}$ by property (2) in Lemma 5.4. In steps (6) and (7), we compose $\texttt{c}_{\texttt{injp}}$ and `ro` by their transitivity. In step (8), we commute semantic invariants by Lemma 5.7. Finally, we merge $\texttt{c}_{\texttt{injp}}$ and $\texttt{CL}\cdot\texttt{LM}\cdot\texttt{MA}$ into `CAinjp` in step (9).

6 END-TO-END VERIFICATION OF HETEROGENEOUS MODULES

In this section, we give a formal account of end-to-end verification of heterogeneous modules based on direct refinements. The discussion focuses on the running example in Fig. 4 and its variants. Additional examples can be found in the technical report [Zhang et al. 2023b].

6.1 Refinement for the Hand-written Server

We use `server_opt.s` instead of `server.s` to illustrate how optimizations are enabled by `ro`. The proof for the unoptimized server is similar with only minor adjustments. A formal definition of LTS for L_S is given below and its transition diagram is given in Fig. 19a.

Definition 6.1. LTS of L_S:

$$\begin{array}{ll}
S_S & := \{\texttt{Calle}\ i\ v_f\ m, \texttt{Callp}\ sp\ v_f\ m, \texttt{Retp}\ sp\ m, \texttt{Rete}\ m\};\\
I_S & := \{(\texttt{Vptr}(b_e,0)[\texttt{int} \to \texttt{ptr} \to \texttt{void}]([i,v_f])@m, \texttt{Calle}\ i\ v_f\ m)\};\\
\to_S & := \{(\texttt{Calle}\ i\ v_f\ m, \texttt{Callp}\ sp\ v_f\ m'') \mid (m',sp) = \texttt{alloc}\ m\ 0\ 8\ \wedge\\
& \quad m'' = m'[sp \leftarrow (i\ \texttt{XOR}\ m[b_k])]\} \cup \{(\texttt{Retp}\ sp\ m, \texttt{Rete}\ m') \mid m' = \texttt{free}\ m\ sp\};\\
X_S & := \{(\texttt{Callp}\ sp\ \texttt{Vptr}(b_p,0)\ m, \texttt{Vptr}(b_p,0)[\texttt{ptr} \to \texttt{void}]([\texttt{Vptr}(sp,0)])@m)\};\\
Y_S & := \{(\texttt{Callp}\ sp\ v_f\ m, res@m', \texttt{Retp}\ sp\ m')\};\\
F_S & := \{(\texttt{Rete}\ m, \texttt{Vundef}@m)\}.
\end{array}$$

The LTS has four internal states as depicted in Fig. 19a. Initialization is encoded in I_S. If the incoming query q^I contains a function pointer $\texttt{Vptr}(b_e, 0)$ which points to `encrypt`, L_S enters $\texttt{Calle}\ i\ v_f\ m$ where i and v_f are its arguments. The first internal transition allocates the stack frame sp and stores the result of encryption $i\ \texttt{XOR}\ m[b_k]$ in sp where b_k contains `key`. Then, it enters `Callp`

$$\begin{tikzcd}
q_C^I \arrow[r, "I_S"] \arrow[d, leftrightarrow, "\mathbb{S}^q"] & \texttt{Calle}\ i\ v_f\ m \arrow[r, "\texttt{alloc}\ sp", "\texttt{store}\ sp"'] \arrow[d, Leftrightarrow, "R"] & \texttt{Callp}\ sp\ v_f\ m_1 \arrow[r, "X_S"] \arrow[d, Leftrightarrow, "R"'] & q_C^O \arrow[r, dashed] \arrow[d, Leftrightarrow, "\mathbb{S}^q"'] & r_C^O \arrow[r, "Y_S"] \arrow[d, leftrightarrow, "\mathbb{S}^r"] & \texttt{Retp}\ sp\ m_2 \arrow[r, "\texttt{free}\ sp"] \arrow[d, Leftrightarrow, "R"] & \texttt{Rete}\ m_3 \arrow[r, "F_S"] \arrow[d, Leftrightarrow, "R"'] & r_C^I \arrow[d, Leftrightarrow, "\mathbb{S}^r"'] \\
q_{\mathcal{A}}^I \arrow[r, "I_{\mathcal{A}}"] & rs@tm \arrow[r, "\texttt{Pallocframe}", "\texttt{...\ Pcall}"'] & rs_1@tm_1 \arrow[r, "X_{\mathcal{A}}"] & q_{\mathcal{A}}^O \arrow[r, dashed] & r_{\mathcal{A}}^O \arrow[r, "Y_{\mathcal{A}}"] & rs_2@tm_2 \arrow[r, "\texttt{Pfreeframe}", "\texttt{Pret}"'] & rs_3@tm_3 \arrow[r, "F_{\mathcal{A}}"] & r_{\mathcal{A}}^I
\end{tikzcd}$$

Fig. 20. Open Simulation between the Optimized Server and its Specification

which is the state before calling process. If the pointer $v_f = \texttt{Vptr}(b_p, 0)$ of the current state points to an external function, L_S issues an outgoing C query q^O with a pointer to its stack frame as its argument. After the external call, Y_S updates the memory with the reply and enters Retp. The second internal transition frees sp and enters Rete and finally returns. Note that complete semantics of L_S is accompanied by a local symbol table which determines the initial value of global variables (key) and asserts that it is a constant. The only difference between the specifications for server_opt.s and server.s is whether key is a constant in the symbol table. The semantics of assembly module [[server_opt.s]] is given by CompCertO whose transition diagram is shown in Fig. 19b. All the states, including queries and replies, are composed of register sets and memories.

Now, we need to prove the following forward simulation. The most important points of the proof are how ro enables optimizations and how injp preserves memory across external calls.

Theorem 6.2. $L_S \leqslant_{\mathbb{C}}$ [[server_opt.s]].

At the top level, we expand $\mathbb{C}$ to $\texttt{ro} \cdot \texttt{wt} \cdot \texttt{CAinjp} \cdot \texttt{asm}_{\texttt{injp}}$ and switch the order of ro and wt by Lemma 5.7. By the vertical compositionality (Theorem 3.4), we first establish $L_S \leqslant_{\texttt{wt}} L_S$ with the well-typed outgoing arguments and return value. [[server_opt.s]] $\leqslant_{\texttt{asm}_{\texttt{injp}}}$ [[server_opt.s]] is proved by Theorem 5.8.

We are left with proving $L_S \leqslant_{\texttt{ro}\cdot\texttt{CAinjp}}$ [[server_opt.s]]. That is, we need to show the simulation diagram in Fig. 20 holds where $\mathbb{S} = \texttt{ro} \cdot \texttt{CAinjp}$ (which mirrors Fig. 6). Here, the given assumptions and the conclusions to be proved are represented as black and red arrows, respectively. For the proof, we need an invariant $R \in \mathcal{K}_{W_{\texttt{ro}\cdot\texttt{CAinjp}}}(S_S, \texttt{regset} \times \texttt{mem})$. The most important point is that ro and injp play essential roles in establishing the invariant. First, ro-valid is propagated from the initial source query q_C^I to internal program states. This guarantees that the value of key read from the source memory states is always 42, hence matching the constant in Pxori 42 RDI in server_opt.s. Second, injp is essential for deriving that memory locations in the target stack frame with offset o ($o < 8$ or $16 \leq o$) are unchanged since they are designated out-of-reach by R. Therefore, the private stack values of the server are protected. For the unoptimized server, the only difference is that we decompose $L_S \leqslant_{\texttt{ro}\cdot\texttt{CAinjp}}$ [[server.s]] into $L_S \leqslant_{\texttt{ro}} L_S$ which trivially holds and $L_S \leqslant_{\texttt{CAinjp}}$ [[server.s]] which can be proved without the help of ro.

6.2 End-to-end Correctness Theorem

We first prove the following source-level refinement where L_{CS} is the top-level specification. Its proof follows the same pattern as Theorem 6.2 but is considerably simpler because the source and target semantics share the same C interface.

Lemma 6.3. $L_{CS} \leqslant_{\texttt{ro}\cdot\texttt{wt}\cdot\texttt{c}_{\texttt{injp}}}$ [[client.c]] $\oplus L_S$.

We then prove the following simulation, which is immediate from the full compositionality, the adequacy for assembly described in §3.1.2, Theorem 5.9, and Theorem 6.2:

Lemma 6.4. [[client.c]] $\oplus L_S \leqslant_{\mathbb{C}}$ [[client.s + server_opt.s]].

```c
/* client.c */
#define N 10
int input[N] = {...};
int result[N];
int i;
void encrypt(int i,
        void(*p)(int*));
```

```c
void request(int *r) {
  if (i == 0) encrypt(input[i++], request);
  else if (0 < i && i < N) {
    result[i-1] = *r;
    encrypt(input[i++], request);
  } else result[i-1] = *r;
}
```

Fig. 21. Client with Multiple Encryption Requests

For end-to-end direct refinement, we need to absorb Lemma 6.3 into Lemma 6.4. The following theorem is easily derived by applying Lemmas 5.5, 5.6 and 5.7.

Lemma 6.5. $\mathbb{C} \equiv \texttt{ro} \cdot \texttt{wt} \cdot \texttt{c}_{\texttt{injp}} \cdot \mathbb{C}$.

The final end-to-end simulation is immediate by vertically composing Lemma 6.3, Lemma 6.4 and refining the simulation convention using Lemma 6.5.

Theorem 6.6. $L_{CS} \leqslant_{\mathbb{C}} [\![\texttt{client.s} + \texttt{server\_opt.s}]\!]$.

6.3 Verification of the Mutually Recursive Client and Server

We introduce a variant of the running example with mutual recursion in Fig. 21. The server remains the same while the client is changed. `request` itself is passed as a callback function to `encrypt`, resulting in recursive calls to `encrypt` for encrypting and storing an array of values. To perform the same end-to-end verification for this example, we only need to define a new top-level specification $L_{CS}{}'$ and prove $L_{CS}{}' \leqslant_{\texttt{ro}\cdot\texttt{wt}\cdot\texttt{c}_{\texttt{injp}}} [\![\texttt{client.c}]\!] \oplus L_S$. Other proofs are either unchanged (e.g., the refinement of the server) or can be derived from Theorem 5.9, full compositionality and adequacy. The complete proofs can be found in the technical report [Zhang et al. 2023b].

7 GENERALITY AND LIMITATIONS OF OUR APPROACH

We explain how our approach may be generalized to support other memory models, compilers and optimizations for first-order languages. We also discuss the limitations of our approach.

7.1 Supporting Different Memory Models and Compilers

At a high level, `injp` is simply a general and transitive relation on evolving *functional memory invariants* (represented as injections) enhanced with *memory protection* to guard against modification to private memory by external calls. Many other first-order memory models can be viewed as employing either a richer or a simplified version of injection as memory invariants and equipped with a similar notion of memory protection. For example, the memory model of CompCertS [Besson et al. 2015] extends injections to map *symbolic values*. The *memory refinements* in the CH_2O memory model [Krebbers 2016] function like injections except that pointer offsets are represented as abstract *paths* pointing into aggregated data structures. The memory model defined by Kang et al. [2015] explicitly divides a memory state into public and private memory. Its memory invariant is an equivalence relation between public source and target memory which is essentially an identity injection. Therefore, uniform KMRs may be defined for those memory models as variants of `injp`.

To prove the transitivity of these KMRs, the key is the construction of interpolating memory states after external calls as described in §4.2. This construction is based on the following general ideas: *1)* as KMRs are transitively composed, more memory gets protected, *2)* the private memory should be identical to the initial memory, and *3)* the public memory should be projected from the updated source memory via memory invariants. As we can see, these ideas are applicable to any memory model with functional memory invariants and a notion of private memory. Therefore, our approach should work for the aforementioned memory models and compilers based on them.

7.2 Supporting Additional Optimization Passes

Given any new optimization pass whose additional rely-guarantee condition can be represented as a semantics invariant I, we may piggyback I onto an enriched `injp` to achieve direct refinement. To see that, note that any I consists of two parts: a condition for initial queries (e.g., `ro-valid` in `ro`) and a condition for replies (e.g., `mem-acc` in `ro`). By extending `injp` to include the latter (just like that Definition 3.3 includes `mem-acc`), if the enriched `injp` is still transitive, then we can easily prove the following proposition which is generalized from Lemma 5.6.

Proposition 7.1. *For any* $I : C \Leftrightarrow C$ *and enriched* `injp`, $I \cdot \mathtt{c}_{\mathtt{injp}} \equiv I \cdot \mathtt{c}_{\mathtt{injp}} \cdot I \cdot \mathtt{c}_{\mathtt{injp}}$.

The proof follows exactly the steps for proving Lemma 5.6. It is based on the two observations we made near the end of §5.2.3, i.e., *1)* the properties for initial queries of I hold along with copying of memory states, and *2)* the properties for replies trivially hold as they are part of the enriched `injp`.

7.3 Limitations

We discuss the limitations of our approach and possible solutions. First, it does not yet support behavior refinements for whole programs in CompCert [Leroy 2023]. This is a technical limitation and can be solved by reducing open simulations into closed simulations in CompCert. Second, the proof for `Unusedglob` assumes that global symbols for removed definitions are preserved as CompCertO's simulation framework requires the same set of global symbols throughout compilation. We need to weaken this requirement to enable removal of global symbols by compilation. Third, open simulations assume given *any* input injection j, the execution outputs *some* injection j' related to j by `injp`. This may not work for memory models with fixed injection functions [Wang et al. 2022]. A possible solution is to enrich `injp` to account for this fixed definition. Finally, given a new optimization, if its rely-guarantee condition cannot be described as a semantic invariant I or if `injp` enriched with I becomes intransitive, then direct refinements may not be derivable. In this case, we may need stronger restrictions on this optimization for our approach to work.

8 EVALUATION AND RELATED WORK

Our Coq development took about 7 person-months and 18.3k lines of code (LOC) on top of CompCertO. We added 3.7k LOC to prove the transitivity of `injp`, 3k LOC to verify the compiler passes as discussed in §5.1, 1.2k LOC for composing simulation conventions as described in the rest of §5 and 7.3k LOC for the Client-Server examples. We also ported CompCertM's example on mutually recursive summation [Song et al. 2020], which adds 3.1k LOC [Zhang et al. 2023b]. For now, the cost of examples is relatively high. However, we observe that a lot of low-level proofs such as pointer arithmetic can be automated by proof scripts, many proofs with predictable patterns can be directly derived from the program structures, and a lot of duplicated lemmas in the examples can be eliminated. We will carry out those exercises in the future which should simplify the proofs significantly. Below we compare our work with other frameworks for VCC and program verification.

8.1 Verified Compositional Compilation for First-Order Languages

In this work, we are concerned with VCC of first-order imperative programs with global memory states and support of pointers. A majority of the work in this setting is based on CompCert. We compare them from the perspectives listed in the first column of Table 2. An answer that is not a simple "Yes" or "No" denotes that special constraints are enforced to support the given feature.

Table 2. Comparison between Work on VCC Based on CompCert

	CompComp	CompCertM	CompCertO	CompCertX	**This Work**
Direct Refinement	No	No	No	No	**Yes**
Vertical Composition	Yes	RUSC	Trivial	CAL	**Yes**
Horizontal Composition	Yes	RUSC	Yes	CAL	**Yes**
Adequacy	No	Yes	Yes	Yes	**Yes**
End-to-end Verification	No	Yes	Unknown	CAL	**Yes**
Free-form Heterogeneity	Yes	Yes	Yes	No	**Yes**
Behavior Refinement	No	Yes	No	Yes	**No**

Compositional CompCert. CompComp supports VCC based on *interaction semantics* which is a specialized version of open semantics with C interfaces [Stewart et al. 2015]. We have already talked about its merits and limitations in §1.2. It is interesting to note that CompComp can also be obtained based on our approach by adopting $\mathtt{c}_{\mathtt{injp}}$ for every compiler pass and exploiting the transitivity of $\mathtt{c}_{\mathtt{injp}}$, which does not require the instrumentation of semantics in CompComp.

CompCertM. CompCertM supports adequacy and end-to-end verification of mixed C and assembly programs. A distinguishing feature of CompCertM is Refinement Under Self-related Contexts or RUSC [Song et al. 2020]. A RUSC relation is a *fixed* collection of simulation relations. By exploiting contexts that are self-relating under all of these simulation relations, horizontal and vertical compositionality are achieved. However, refinements based on RUSC relations can be difficult to use as they are not extensional. For example, the complete open refinement relation $\leqslant_{R_1+\ldots+R_9}$ in CompCertM carries 9 RUSC relations $R_1, \ldots, R_9$ (6 for compiler passes and 3 for source-level verification). To establish the refinement between `a.s` and its specification L_S, one needs to prove L_S are self-simulating over *all* 9 simulation relations. This can quickly get out of hand as more modules and more compiler passes are introduced. By contrast, we only need to prove direct refinement *for once* and the refinement is open to further horizontal or vertical composition. On the other hand, CompCertM supports behavior refinement of closed programs which we do not yet (See §7.3).

CompCertO. Vertical composition is a trivial pairing of simulations in CompCertO, which exposes internal compilation steps. CompCertO tries to alleviate this problem via ad-hoc refinement of simulation conventions. The resulting top-level convention is $\mathbb{C}_{\mathtt{CCO}} = \mathcal{R}^* \cdot \mathtt{wt} \cdot \mathtt{CL} \cdot \mathtt{LM} \cdot \mathtt{MA} \cdot \mathtt{asm}_{\mathtt{vainj}}$ where $\mathcal{R} = \mathtt{c}_{\mathtt{injp}} + \mathtt{c}_{\mathtt{inj}} + \mathtt{c}_{\mathtt{ext}} + \mathtt{c}_{\mathtt{vainj}} + \mathtt{c}_{\mathtt{vaext}}$ is a sum of conventions parameterized over KMRs. In particular, $\mathtt{c}_{\mathtt{vaext}}$ is an ad-hoc combination of KMR and internal invariants for optimizations. $\mathcal{R}^*$ means that $\mathcal{R}$ may be repeated for an arbitrary number of times. Since the top-level summation of KMRs is similar to that in CompCertM, we need to go through a reasoning process similar to CompCertM, only more complicated because of the need to reason about internal invariants of optimizations in $\mathtt{c}_{\mathtt{vaext}}$ and indefinitely repeated combination of all the KMRs by $\mathcal{R}^*$. Therefore, it is unknown if the correctness theorem of CompCertO suffices for end-to-end program verification.

CompCertX. CompCertX [Gu et al. 2015; Wang et al. 2019] realizes a weaker form of VCC that only allows assembly contexts to invoke C programs, but not the other way around. Therefore, it does not support horizontal composition of modules with mutual recursions. The compositionality and program verification are delegated to Certified Abstraction Layers (CAL) [Gu et al. 2015, 2018]. Furthermore, CompCertX does not support stack-allocated data (e.g., our server example). However, its top-level semantic interface is similar to our interface, albeit not carrying a symmetric rely-guarantee condition. This indicates that our work is a natural evolution of CompCertX.

VCC for Concurrent Programs. VCC for concurrent programs needs to deal with multiple threads and their linking. CASCompCert is an extension of CompComp that supports compositional compilation of concurrency with no (or benign) data races [Jiang et al. 2019]. To make CompComp's approach to VCC work in a concurrent setting, CASCompCert imposes some restrictions including not supporting stack-allocated data and allowing only nondeterminism in scheduling threads. A recent advancement based on CASCompCert is about verifying concurrent programs [Zha et al. 2022] running on weak memory models using the promising semantics [Kang et al. 2017; Lee et al. 2020]. We believe the ideas in CASCompCert are complementary to this work and can be combined with our approach to achieve VCC for concurrency with cleaner interface and less restrictions.

8.2 Verified Compositional Compilation for Higher-Order Languages

Another class of work on VCC focuses on compilation of higher-order languages. In this setting, the main difficulty comes from complex language features together with higher-order states. A prominent example is the Pilsner compiler [Neis et al. 2015] that compiles a higher-order language into some form of assembly programs. The technique Pilsner adopts is called *parametric simulations* that evolves from earlier work on reasoning about program equivalence via bisimulation [Hur et al. 2012a]. Another line of work is multi-language semantics [Patterson and Ahmed 2019; Patterson et al. 2017; Perconti and Ahmed 2014; Scherer et al. 2018] where a language combining all source, intermediate and target languages is used to formalize semantics. Compiler correctness is stated as contextual equivalence or logical relations. It seems that our techniques are not directly applicable to those work because relations on higher-order states cannot deterministically fix the interpolating states. A possible solution is to divide the higher-order memory into a first-order and a higher-order part such that the former does not contain pointers to the latter (forming a closure). By encapsulating higher-order programs inside first-order states, we may be able to apply our approach.

The high-level ideas for constructing interpolating states for proving transitivity of `injp` can also be found in some of the work on program equivalence [Ahmed 2006; Hur et al. 2012b]. To the best of our knowledge, our approach is the first concrete implementation of these ideas that works for a realistic optimizing compiler for imperative languages with non-trivial memory models.

8.3 Frameworks for Compositional Program Verification

Researchers have proposed frameworks for compositional program verification based on novel semantics, refinements and separation logics [Chappe et al. 2023; Gu et al. 2015, 2018; He et al. 2021; Sammler et al. 2023; Song et al. 2023; Xia et al. 2019]. These frameworks aim at broader program verification and may be combined with our approach to generate more flexible end-to-end verification techniques. For example, to support more flexible certified abstraction layers, we may combine our approach with data abstraction in CAL and extend horizontal linking to work with abstraction layers. More details can be found in the technical report [Zhang et al. 2023b].

9 CONCLUSION AND FUTURE WORK

We have proposed an approach to compositional compiler correctness for first-order languages via direct refinements between source and target semantics at their native interfaces, which overcomes the limitations of the existing approaches on compositionality, adequacy and other important criteria for VCC. In the future, we plan to support behavior (trace) refinement for closed programs by reducing our open simulation into the whole-program correctness theorem for the original CompCert. We also plan to combine our work with refinement-based program verification like certified abstraction layers to support more substantial applications. Another research direction is to apply our approach to different memory models and compilers for first-order and higher-order languages, which will better test the limit of our approach and the usefulness of our discoveries.

DATA-AVAILABILITY STATEMENT

The Coq artifact containing the formal developments described in this paper is available on Zenodo [Zhang et al. 2023a]. A technical report of this work is available on arXiv [Zhang et al. 2023b].

ACKNOWLEDGMENTS

We would like to thank our shepherd Yannick Zakowski and the anonymous referees for their helpful feedback which improved this paper significantly. This work is supported in part by the National Natural Science Foundation of China (NSFC) under Grant No. 62002217 and 62372290, and by the Natural Science Foundation of the United States (NSF) under Grant No. 1763399, 2019285, and 2313433. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

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. https://doi.org/10.1007/11693024_6

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2015. A Concrete Memory Model for CompCert. In *Proc. 6th Interactive Theorem Proving (ITP'15) (LNCS, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, Cham, 67–83. https://doi.org/10.1007/978-3-319-22102-1_5

Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proc. ACM Program. Lang.* 7, POPL, Article 61 (January 2023), 31 pages. https://doi.org/10.1145/3571254

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. https://doi.org/10.1145/2775051.2676975

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjober, Hao Chen, David Costanzo, and Tahnia Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proc. 2018 ACM Conference on Programming Language Design and Implementation (PLDI'18)*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, New York, NY, USA, 646–661. https://doi.org/10.1145/3192366.3192381

Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Ştefănescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. 2021. A Type System for Extracting Functional Specifications from Memory-Safe Imperative Programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 135 (October 2021), 29 pages. https://doi.org/10.1145/3485512

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012a. 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. https://doi.org/10.1145/2103656.2103666

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2012b. *The Transitive Composability of Relation Transition Systems.* Technical Report, MPI-SWS-2012-002. MPI-SWS. https://www.mpi-sws.org/tr/2012-002.pdf

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. https://doi.org/10.1145/3314221.3314595

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proc. 44th ACM Symposium on Principles of Programming Languages (POPL'17)*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A Formal C Memory Model Supporting Integer-Pointer Casts. In *Proc. 2015 ACM Conference on Programming Language Design and Implementation (PLDI'15)*, David Grove and Stephen M. Blackburn (Eds.). ACM, New York, NY, USA, 326–335. https://doi.org/10.1145/2737924.2738005

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. https://doi.org/10.1145/3453483.3454097

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

Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *Proc. 2020 ACM Conference on Programming Language Design and Implementation (PLDI'20).* ACM, New York, NY, USA, 362–376. https://doi.org/10.1145/3385412.3386010

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

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2.* Research Report RR-7987. INRIA. 26 pages. https://hal.inria.fr/hal-00703441

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. https://doi.org/10.1145/2784731.2784764

Daniel Patterson and Amal Ahmed. 2019. The Next 700 Compiler Correctness Theorems (Functional Pearl). *Proc. ACM Program. Lang.* 3, ICFP, Article 85 (August 2019), 29 pages. https://doi.org/10.1145/3341689

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: Reasonably Mixing a Functional Language with Assembly. *SIGPLAN Not.* 52, 6 (2017), 495–509. https://doi.org/10.1145/3140587.3062347

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *Proc. 23rd European Symposium on Programming (ESOP'14) (LNCS, Vol. 8410)*, Zhong Shao (Ed.). Springer, Cham, 128–148. https://doi.org/10.1007/978-3-642-54833-8_8

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. https://doi.org/10.1145/3571220

Gabriel Scherer, Max New, Nick Rioux, and Amal Ahmed. 2018. Fabous Interoperability for ML and a Linear Language. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer, Cham, 146–162. https://doi.org/10.1007/978-3-319-89366-2_8

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. https://doi.org/10.1145/3371091

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

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. https://doi.org/10.1145/2676726.2676985

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. https://doi.org/10.1145/3290375

Yuting Wang, Ling Zhang, Zhong Shao, and Jérémie Koenig. 2022. Verified Compilation of C Programs with a Nominal Memory Model. *Proc. ACM Program. Lang.* 6, POPL, Article 25 (January 2022), 31 pages. https://doi.org/10.1145/3498686

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

Junpeng Zha, Hongjin Liang, and Xinyu Feng. 2022. Verifying Optimizations of Concurrent Programs in the Promising Semantics. In *Proc. 2021 ACM Conference on Programming Language Design and Implementation (PLDI'22).* ACM, New York, NY, USA, 903–917. https://doi.org/10.1145/3519939.3523734

Ling Zhang, Yuting Wang, Jinhua Wu, Jérémie Koenig, and Zhong Shao. 2023a. Fully Composable and Adequate Verified Compilation with Direct Refinements between Open Modules (Artifact). https://doi.org/10.5281/zenodo.10036618

Ling Zhang, Yuting Wang, Jinhua Wu, Jérémie Koenig, and Zhong Shao. 2023b. Fully Composable and Adequate Verified Compilation with Direct Refinements between Open Modules (Technical Report). https://doi.org/10.48550/arXiv.2302.12990

Received 2023-07-11; accepted 2023-11-07