# A Compositional Semantics for Verified Separate Compilation and Linking

Tahina Ramananandro Zhong Shao Shu-Chun Weng Jérémie Koenig Yuchen Fu[1]

Yale University [1]Massachusetts Institute of Technology

Technical Report YALEU/DCS/TR-1494, December 2014

## Abstract

Recent ground-breaking efforts such as CompCert have made a convincing case that mechanized verification of the compiler correctness for realistic C programs is both viable and practical. Unfortunately, existing verified compilers can only handle whole programs—this severely limits their applicability and prevents the linking of verified C programs with verified external libraries. In this paper, we present a novel compositional semantics for reasoning about open modules and for supporting verified separate compilation and linking. More specifically, we replace external function calls with explicit events in the behavioral semantics. We then develop a verified linking operator that makes lazy substitutions on (potentially reacting) behaviors by replacing each external function call event with a behavior simulating the requested function. Finally, we show how our new semantics can be applied to build a refinement infrastructure that supports both vertical composition and horizontal composition.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs, formal methods; D.3.4 [*Programming Languages*]: Processors—Compilers; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

***Keywords*** Compositional Semantics; Vertical and Horizontal Composition; Verified Compilation and Linking.

## 1. Introduction

Compiler verification has long been considered as a theoretically deep and practically important research subject. It addresses the very question of program equivalence (or simulation), a primary reason that we need to define formal semantics for programming languages. It is important for practical software developers since compiler bugs can lead to the silent generation of incorrect programs, which could lead to unexpected crashes and security holes.

Recent work on CompCert [13, 12] has shown that mechanized verification of the compiler correctness for C is both viable and practical, and the resulting compiler is indeed empirically much more reliable than traditional (unverified) ones [23]. The success of CompCert can be partly attributed to its uses of simple (small-step and/or big-step) operational semantics [15], a shared behavioral specification language (capable of describing terminating, stuck, silently diverging, and reacting behaviors), and a unified C memory model [14] for all of its compiler intermediate languages. The simplicity of the CompCert semantics made it possible and practical to mechanically verify the correctness of many compilation phases under a reasonable amount of effort.

One important weakness of CompCert is that it can only handle whole programs. This severely limits its applicability. A computer program is often not just a single piece of code written and compiled at once, but is instead obtained by compiling and linking different *modules*, or *compilation units*, that can be originally written in different programming languages, independently of each other. From the compilation point of view, the final program is obtained by linking different *object files*, each of which is either written directly or obtained by compiling a source compilation unit. Different compilers can be used for different modules.

From the program-verification point of view, a computer program is almost never verified as a whole, but for each compilation unit, its source code (or object file, if written directly) is verified independently from the implementation of the other modules. Without support for separate compilation and linking, verified C programs, even if correctly compiled by CompCert, cannot be linked with verified external libraries.

An open problem for supporting verified separate compilation and linking is to find a simple *compositional* semantics for open modules and to specify and reason about such semantic behaviors in a language-independent way. Following Hur *et al* [10, 11], we want to achieve compositionality in the two dimensions:

- *vertical composition* corresponds to successive compilation passes on a given compilation unit. Each compilation pass can be an optimization to make a program more efficient while staying at the same representation level, or a compilation phase from one intermediate representation to another: how to define compositional semantics of intermediate programs in a language-independent format so that we can show that each compilation pass does not introduce unwanted behaviors?
- *horizontal composition* corresponds to the linking of different modules at the same level (i.e. at the level of object files, or at the same intermediate level). It corresponds to the notion of *program composition*: local reasoning shall allow studying the behavior of program components when placed in an abstractly specified context. But conversely, when linking them together, compilation units will play the role of contexts for other modules. More generally, this notion becomes symmetric when they can mutually call functions in each other.

In this paper, we present a novel compositional semantics (for open modules) that supports both vertical composition and horizontal composition for C-like languages. Traditionally, operational semantics focuses on reasoning about the behaviors of a whole program. This partly explains why CompCert does not handle open modules. A significant attempt toward developing compositional semantics has been denotational semantics, and the underlying domain theory has led to a wide body of research; however, denotational models become difficult to extend as we add more language features and they are harder to mechanize in a proof assistant.

Our paper makes the following contributions:

- We develop a *compositional semantics* (denoted as $[\![\cdot]\!]_{\mathsf{comp}}$, see Sec. 4) to help reason about open modules. Our key idea is to model external function calls in a similar way as how compositional semantics for concurrent languages [5] models environmental transitions. The behavior of a call to an external function $f$ is modeled as an event $\texttt{Extcall}(f, m, m')$, with $m$ and $m'$ denoting memory states before and after the call. A function body that makes $n$ consecutive external calls can be modeled as a sequence of event traces of the form $\texttt{Extcall}(f_1, m_1, m_1') :: \texttt{Extcall}(f_2, m_2, m_2') :: \ldots :: \texttt{Extcall}(f_n, m_n, m_n')$, with the assumption that segments between two external call events, e.g., $(m_1', m_2)$ and $(m_{n-1}', m_n)$, are transitions made by the function body itself. We show how to extend the CompCert-style behavioral semantics with these new external call events and how to use a shared behavioral specification language (as in CompCert) to support vertical compositionality.
- We develop a *linking* operator directly at the semantic level (denoted as $\bowtie$, see Sec. 5), based on a resolution operator which makes a lazy substitution on behaviors by replacing each external function call event with a behavior simulating the requested function. We show that applying the linking operator to the compositional semantic objects ($\psi_1$ and $\psi_2$ for open modules $\mathfrak{u}_1$ and $\mathfrak{u}_2$) will yield the same compositional semantic object ($\psi_1 \bowtie \psi_2$) for the linked module ($\mathfrak{u}_1 \uplus \mathfrak{u}_2$). Since linking is directly done on semantic objects, our approach can also be applied to components compiled from different source languages; for example, a module $\mathfrak{u}_\alpha$ (in language A) can be compiled by compiler $C_\alpha$ and linked with another module $\mathfrak{u}_\beta$ compiled by compiler $C_\beta$, yielding a resulting binary with the semantic object $[\![C_\alpha(\mathfrak{u}_\alpha)]\!]_{\mathsf{comp}} \bowtie [\![C_\beta(\mathfrak{u}_\beta)]\!]_{\mathsf{comp}}$.
- Thanks to this new compositional semantics and semantic linking, we develop a refinement infrastructure (denoted as $\sqsubseteq$, see Sec. 6) that unifies program verification and verified separate compilation: each verification step, as well as each compilation step, is actually a refinement step. The transitivity property of our refinement relation implies vertical composition; and the congruence property (a.k.a. monotonicity, see Theorem 2) implies horizontal composition.
- Unlike the CompCert whole program semantics, which does not expose memory states in its event traces, compositional semantics for open modules may make part of the memory state observable (e.g., as in an external call event $\texttt{Extcall}(f, m, m')$). This creates challenges for verifying compilation phases that alter memory states. We introduce $\alpha$-refinement (denoted as $\sqsubseteq_\alpha$, see Sec. 7), a generalization of $\sqsubseteq$ between the source and the target memory states. We show how $\alpha$-refinement can be used to verify the correctness of the memory-changing phases in CompCert, and we have successfully reimplemented (and verified in Coq) the Clight-to-Cminor phase—the CompCert pass that uses the most sophisticated memory injection relation—using $\alpha$-like memory bijection.

All our proofs have been carried out in Coq [21] and can be found at the companion web site [19]. The implementation includes the generic compositional semantics and linking framework, an instantiation of the framework for the common subexpression elimination pass, and a new implementation of the CompCert memory model with block tags and the Clight-to-Cminor compilation phase using memory bijection.

## 2. Preliminary: small-step and big-step semantics

In this section, we define the general notion of small-step semantics, or transition systems, and explain how to automatically construct big-step semantics based on them. Throughout the paper, when we define a small-step semantics, we always construct the corresponding big-step semantics based on this section.

Small-step semantics illustrates how to execute programs with minimal steps. Big-step semantics gives us the meaning of programs as a whole. When studying the meaning of a program, we focus not only on whether it terminates or diverges, but also on its interaction with the outside environment through *events* like input and output, network communications, etc. We borrow all these definitions from the CompCert verified compiler [12].

Before diving into semantics, we first go through some notations on sets, (finite) lists, and (infinite) streams. We use $X^?$ to denote the set of all subsets of $X$ with 0 or 1 element. For any subset $Y \subseteq X^?$, we liberally write $x \in Y$ instead of $\{x\} \in Y$. The standard notation for power set $\mathcal{P}(X)$ is also used.

For any set $X$, $X^*$ denotes the set of finite lists of elements of $X$. Such lists can be either empty ($\varepsilon$) or nonempty ($x :: l$). For two lists $l_1, l_2 \in X^*$, $l_1 +\!\!+ l_2$ is their concatenation. $X^\infty$ denotes the set of infinite streams of $X$, which are defined coinductively such that all elements are of the form $x :::: \mathfrak{l}$ where $x \in X$ and $\mathfrak{l} \in X^\infty$. The coinductive definition allows (actually, requires) streams to be infinite, in contrast to lists, which are defined inductively and must be finite. Prepending a list $l$ of $X$ in front of a stream $\mathfrak{l}$ of $X$ is written $l +\!\!\!+\!\!\!+ \mathfrak{l}$. We write $\mathfrak{l}_1 \sim \mathfrak{l}_2$ meaning two streams are bisimilar (coinductively, $\exists x, \forall i = 1, 2, \exists \mathfrak{l}_i', \mathfrak{l}_i = x :::: \mathfrak{l}_i'$ and $\mathfrak{l}_1' \sim \mathfrak{l}_2'$).

**Definition 1** (Small-step semantics)**.** *A* small-step semantics *(or a* transition system*) is a tuple* $\mathfrak{S} = (\mathcal{E}, \mathcal{S}, \rightarrow, \mathcal{R}, \mathcal{F})$ *where:*

- $\mathcal{E}$ *is the set of* events*.*
- $\mathcal{S}$ *is the set of* configurations *(or* states*).*
- $(\rightarrow) \subseteq \mathcal{S} \times \mathcal{E}^? \times \mathcal{S}$ *is the* transition relation*, usually written in infix forms* $s \xrightarrow{e} s'$ *and* $s \rightarrow s'$*. We say that* $s$ *makes one step (or transition) to* $s'$*, producing an event* $e$ *(if any). A step producing no event is* silent*.*
- $\mathcal{R}$ *is the set of* results*.*
- $\mathcal{F} \subseteq (\mathcal{S} \times \mathcal{R})$ *is a relation associating* final states *with* results*. A configuration* $s$ *is said to be final with result* $r$ *if, and only if,* $(s, r) \in \mathcal{F}$*.*

The transition relation may be *nondeterministic*: for a given configuration $s$, there can be several possible configurations $s'$ such that $s \rightarrow s'$ (or $s \xrightarrow{e} s'$ for some event $e$).

Then, a configuration $s$ can make several transitions to $s'$ producing a finite list $\sigma$ of events in $\mathcal{E}$, which we write $s \xrightarrow{\sigma}{}^* s'$ (or $s \xrightarrow{\sigma}{}^+ s'$ if there is at least one step) and define as the reflexive-transitive (resp. transitive) closure of the transition step relation:

$$\frac{s \rightarrow s'}{s \xrightarrow{\varepsilon}{}^+ s'} \qquad \frac{s \xrightarrow{e} s'}{s \xrightarrow{e::\varepsilon}{}^+ s'} \qquad \frac{s \xrightarrow{\sigma_1}{}^+ s_1 \quad s_1 \xrightarrow{\sigma_2}{}^+ s_2}{s \xrightarrow{\sigma_1 +\!\!+ \sigma_2}{}^+ s_2} \qquad \frac{}{s \xrightarrow{\varepsilon}{}^* s} \qquad \frac{s \xrightarrow{\sigma}{}^+ s'}{s \xrightarrow{\sigma}{}^* s'}$$

We can then define the *behavior* of a transition system from an initial state $s_0 \in \mathcal{S}$.

- It can perform finitely many transition steps to some final configuration $s'$ such that $(s', r') \in \mathcal{F}$ for some $r'$. In this case, we say that it is a *terminating* behavior. For such a behavior, we record the result $r'$ and its *trace* of events, a finite list, produced to go from $s_0$ to $s'$.
- It can perform finitely many transition steps to some non-final configuration $s'$ but from which no step is possible. In this case, we say that it is a *going-wrong* (or *stuck*) behavior, for which we record the trace produced from $s_0$ to $s'$. In practice, $s'$ corresponds to a configuration requesting an invalid operation such as out-of-bounds array access or division by zero.
- It can perform infinitely many transition steps. For such cases, we need to distinguish whether a finite or infinite list of events

is produced during these transition steps. (1) In the finite event case, finitely many steps are performed to some state $s'$ from which infinitely many silent transitions are performed. (2) In the infinite event case, starting from any state, a non-silent transition can always be reached within finitely many steps; we record the trace as an infinite stream of events.

The bullets above are formally defined as follows.

**Definition 2** (Behaviors)**.** *Given a set of events $\mathcal{E}$ and a set of results $\mathcal{R}$, we define the set of behaviors $\mathcal{B}$ as follows:*

| | | | | |
|---|---|---|---|---|
| $b$ | $\in$ | $\mathcal{B}$ | | *Behavior* |
| | $::=$ | $\sigma\downarrow(r)$ | $(\sigma \in \mathcal{E}^*, r \in \mathcal{R})$ | *Terminating behavior* |
| | $\mid$ | $\sigma\lightning$ | $(\sigma \in \mathcal{E}^*)$ | *Going-wrong behavior* |
| | $\mid$ | $\sigma\nearrow$ | $(\sigma \in \mathcal{E}^*)$ | *Diverging behavior* |
| | | | | *(finitely many events, then silently diverges)* |
| | $\mid$ | $\varsigma\nRightarrow$ | $(\varsigma \in \mathcal{E}^\infty)$ | *Reacting behavior* |
| | | | | *(diverging with infinitely many events)* |

*The concatenation of an event $e$ (resp. of an event list $\sigma$) and a behavior $b$ is written $e \cdot b$ (resp. $\sigma \bullet b$) and defined in a straightforward way:*

$$\begin{array}{rclcrcl} e\cdot(\sigma\downarrow(r)) & = & (e::\sigma)\downarrow(r) & \qquad & \varepsilon\bullet b & = & b \\ e\cdot(\sigma\lightning) & = & (e::\sigma)\lightning & & (e::\sigma)\bullet b & = & e\cdot(\sigma\bullet b) \\ e\cdot(\sigma\nearrow) & = & (e::\sigma)\nearrow & & & & \\ e\cdot(\varsigma\nRightarrow) & = & (e::::\varsigma)\nRightarrow & & & & \end{array}$$

**Definition 3** (Stuck, silently diverging, reacting states)**.** *Given a small-step semantics $\mathfrak{S} = (\mathcal{E}, \mathcal{S}, \rightarrow, \mathcal{R}, \mathcal{F})$, a configuration $s$ is:*

- stuck *(written $s\lightning$) if, and only if, there is no $s'$ (resp. and there is no $e \in \mathcal{E}$) such that $s \rightarrow s'$ (resp. $s \xrightarrow{e} s'$)*
- silently diverging *(written $s\nearrow$) if, and only if, coinductively, there is a configuration $s'$ such that $s \rightarrow s'$ and $s'\nearrow$.*
- reacting *with the infinite event stream $\varsigma$ (written $s \nRightarrow \varsigma$) if, and only if, coinductively, there is a* nonempty *finite event list $\sigma$ and a configuration $s'$ such that $s \xrightarrow{\sigma}{}^{+} s'$, and an infinite event stream $\varsigma'$ such that $s' \nRightarrow \varsigma'$ and $\varsigma \sim \sigma \mathbin{⧺} \varsigma'$.*

If the transition relation is nondeterministic, the transition system may have several behaviors from a single initial state. We want to describe [1] the set of all the possible behaviors of the transition system from a given configuration.

**Definition 4** (Big-step semantics)**.** *Given a small-step semantics $\mathfrak{S} = (\mathcal{E}, \mathcal{S}, \rightarrow, \mathcal{R}, \mathcal{F})$, the* big-step *semantics $(\!|\mathfrak{S}|\!)$ of $\mathfrak{S}$ is a function from $\mathcal{S}$ to $\mathcal{P}(\mathcal{B})$ such that, for each configuration $s_0$, $(\!|\mathfrak{S}|\!)(s_0)$ is the set of all possible behaviors from $s_0$, defined as follows:*

$$\begin{array}{rcl} (\!|\mathfrak{S}|\!)(s_0) & = & \{\sigma\downarrow(r) : s_0 \xrightarrow{\sigma}{}^{*} s \wedge (s,r) \in \mathcal{F}\} \\ & \cup & \{\sigma\lightning : s_0 \xrightarrow{\sigma}{}^{*} s \ \wedge\ s\lightning\} \\ & \cup & \{\sigma\nearrow : s_0 \xrightarrow{\sigma}{}^{*} s \ \wedge\ s\nearrow\} \\ & \cup & \{\varsigma\nRightarrow : s_0 \nRightarrow \varsigma\} \end{array}$$

Below are some C examples showing the use of behaviors. The command `printf('a');`, printing the character "a" on screen, produces an observable event `OUT(a)`. The results are `int` values.

```
int main () { printf('a'); return 2; }
```
has the behavior $\texttt{OUT(a)} :: \varepsilon\downarrow(2)$.

```
int main () { printf('a'); 3/0; return 4;}
```
has the behavior $\texttt{OUT(a)} :: \varepsilon\lightning$.

```
int main () { printf('a'); while (1) {} }
```
has the behavior $\texttt{OUT(a)} :: \varepsilon\nearrow$.

```
int main () { while (1) printf('b'); }
```
has the behavior $\texttt{OUT(b)} :::: \ldots :::: \texttt{OUT(b)} :::: \ldots \nRightarrow$

**Lemma 1.** *For any configuration $s_0 \in \mathcal{S}$, the transition system has at least one behavior from $s_0$: $(\!|\mathfrak{S}|\!)(s_0) \neq \emptyset$.*

*Proof.* Done in CompCert [12]. Requires the excluded middle to distinguish whether the program has finite or infinite sequence of steps, and an axiom of constructive indefinite description to construct the infinite event sequence in the reacting case. □

## 3. Starting point: a language with function calls

Our work studies a semantic notion of linking two compilation units at the level of their behaviors, independently of the languages in which they are defined. We first show how to derive a set of behaviors for an open module from a language with function calls.

In this section, we first describe our starting point, the semantics of a language with function calls. For now, we consider only whole programs. Then we will show in Sec. 4 how to make its semantics compositional and suitable for open modules.

Our starting point language makes a memory state evolve throughout the whole program execution across function calls, and a local state (e.g. local variables) evolve within each function call. When a function returns, we consider that its result is the new memory state obtained at the end of the execution of the function, just before it hands over to its caller. Our Coq development also features argument passing and return value, but for the sake of presentation, we do not mention them here. See Sec. 6.5 for more details about our Coq implementation.

The key point of the semantics of our language is that the local state of a function call cannot be changed by other function calls [2]: when a function is called, the local state of the caller is "frozen" until the callee returns.

In this section, we consider the semantics of a *whole program*, which does not contain external function calls. A program consists of several functions. We are interested in the behaviors of each function in the program for all memory states under which the function is called.

A program is modeled as a partial function from function names to code. Let $p$ be a program and $f$ be a function name, $p(f)$ is then the body of $f$. When $f$ is called under a memory state $m$, an initial local state $\mathsf{Init}(p(f), m)$ is first created from the code $p(f)$. The local state and the memory state evolve together by performing *local transition steps* which can produce some events. Eventually, the local state may correspond to a *return state*, meaning that the execution of the function has reached termination. The memory state is the result of the function call.

But a local state $l$ can also correspond to *calling* some other function $f'$: in that case, $l$ is first saved into a *continuation frame* $k = \mathsf{Backup}(m, l)$ that is put on top of a *continuation stack*, then the function $f'$ is called and run. If the execution of this callee reaches a return state, with a new memory state $m'$, then the execution goes back to the caller by retrieving, from the stack, the frame $k$ and constructing a new local state $\mathsf{Restore}(m', k)$.

So, to obtain the behaviors of a function $f$ under a memory state $m$, we just have to big-step such a small-step semantics. Note that when an execution reaches a configuration where the local state is

[1] It would be tempting to *compute* the set of all behaviors. We can argue that the cases described above are exhaustive. However, we do not intend to actually enumerate all possible behaviors of the transition system. In other words, we do not intend to *decide* whether a transition system terminates or diverges. Instead, we describe the set $(\!|\mathfrak{S}|\!)(s_0)$ by defining the meaning of the predicate $b \in (\!|\mathfrak{S}|\!)(s_0)$. This is what happens inside CompCert, and we follow this approach as well in our Coq formalization.

[2] Stack-allocated variables that can have their addresses taken do not belong to the local state. They are allocated in the memory state allowing their addresses to be passed to other functions and their contents to be modified. By contrast, the local state will only store the addresses of such variables. Those addresses shall not be changed by other functions. This is how CompCert models the C stack [4].

a return state and the stack is empty, it means that the function is done executing and there is no caller function to return to, hence it is the final configuration for a function execution. The result of the execution is the memory state of such a configuration.

**Definition 5** (Language with function calls)**.** *A* language with function calls *is a tuple:*

$$\mathfrak{L} = (\mathbf{F}, \mathbf{C}, \mathbf{MS}, \mathbf{LS}, \mathsf{Init}, \mathsf{Kind}, \mathbf{E}, \rightarrow, \mathbf{K}, \mathsf{Backup}, \mathsf{Restore})$$

- **F** *is the set of* function names.
- **C** *is the set of pieces of* code *corresponding to the bodies of functions (i.e., the* syntax *of the language).*
- **MS** *is the set of* memory states.
- **LS** *is the set of* local states.
- $\mathsf{Init} : (\mathbf{C} \times \mathbf{MS}) \longrightarrow \mathbf{LS}$ *is a total function that gives the initial local state when starting to execute a function body.*
- Kind *is a total function such that, for any local state* $l \in \mathbf{LS}$*,* $\mathsf{Kind}(l)$ *may be either:*
  - $\texttt{Call}(f)$ *to say that* $l$ *corresponds to calling a function* $f \in \mathbf{F}$*. Then we define* $\mathbf{LS}_{\texttt{Call}} = \{l : \exists f, \mathsf{Kind}(l) = \texttt{Call}(f)\}$.
  - `Return` *to say that* $l$ *is a return state.*
  - `Normal`*: none of the above. Then, we define* $\mathbf{LS}_{\texttt{Normal}} = \{l : \mathsf{Kind}(l) = \texttt{Normal}\}$.
- **E** *is the set of* events.
- $(\rightarrow) \subseteq ((\mathbf{MS} \times \mathbf{LS}_{\texttt{Normal}}) \times \mathbf{E}^? \times (\mathbf{MS} \times \mathbf{LS}))$ *is the* internal step *relation, usually written in infix forms* $(m, l) \xrightarrow{e} (m', l')$ *and* $(m, l) \rightarrow (m', l')$.
- **K** *is the set of* continuation stack frames.
- $\mathsf{Backup} : (\mathbf{MS} \times \mathbf{LS}_{\texttt{Call}}) \longrightarrow \mathbf{K}$ *is a total function that* saves *the current local state into a stack frame upon function call.*
- $\mathsf{Restore} : (\mathbf{MS} \times \mathbf{K}) \longrightarrow \mathbf{LS}$ *is a total function that* restores *a new local state from a stack frame upon callee return.*

**Definition 6.** *Let* $\mathfrak{L}$ *be a language with function calls. A* program *is a partial function from function names to code.*

**Definition 7** (Procedural semantics)**.** *Let* $\mathfrak{L}$ *be a language with function calls and* $p$ *be a program in* $\mathfrak{L}$*, the* procedural small-step semantics $\mathsf{Proc}[\mathfrak{L}, p]$ *is defined as follows:*

- *The set of events is* **E**.
- *The set of configurations is* $\mathbf{MS} \times \mathbf{LS} \times \mathbf{K}^*$.
- *The transition relation* $(\mathfrak{L}, p) \vdash \cdot \rightarrow \cdot$ *is defined as follows:*

$$\frac{\mathsf{Kind}(l) = \texttt{Normal} \quad (m, l) \rightarrow (m', l')}{(\mathfrak{L}, p) \vdash (m, l, \kappa) \rightarrow (m', l', \kappa)}$$

$$\frac{\mathsf{Kind}(l) = \texttt{Normal} \quad (m, l) \xrightarrow{e} (m', l') \quad e \in \mathbf{E}}{(\mathfrak{L}, p) \vdash (m, l, \kappa) \xrightarrow{e} (m', l', \kappa)}$$

$$\frac{\mathsf{Kind}(l) = \texttt{Call}(f) \quad p(f) = c \quad l' = \mathsf{Init}(c, m) \quad k = \mathsf{Backup}(m, l)}{(\mathfrak{L}, p) \vdash (m, l, \kappa) \rightarrow (m, l', k :: \kappa)}$$

$$\frac{\mathsf{Kind}(l) = \texttt{Return} \quad l' = \mathsf{Restore}(m, k)}{(\mathfrak{L}, p) \vdash (m, l, k :: \kappa) \rightarrow (m, l', \kappa)}$$

- *The set of results is* **MS**.
- *The final configurations with result* $m$ *are the configurations* $(m, l, \varepsilon)$ *where* $\mathsf{Kind}(l) = \texttt{Return}$.

*Let* **B** *be the set of behaviors on events* **E**. *The* procedural big-step semantics *of* $p$ *is the function* $[\![p]\!] : \mathrm{dom}(p) \longrightarrow \mathbf{MS} \longrightarrow \mathcal{P}(\mathbf{B})$ *obtained from big-stepping the procedural small-step semantics:*

$$[\![p]\!](f)(m) = (\!|\mathsf{Proc}[\mathfrak{L}, p]|\!)(m, \mathsf{Init}(p(f), m), \varepsilon)$$

Note that a function call is only triggered and the function name resolved when Kind returns $\texttt{Call}(f)$, which depends solely on local states. It does not matter how the calling request gets put into the local state. Whether it originates from the code, that is, a direct call, or prepared by the caller in the memory state only to be moved to the local state now, which indicates an indirect call, the procedural semantics handles them the same. This means that our setting transparently handles C-style higher-order function pointers, without having to provide a special case for them.

## 4. Compositional semantics

The procedural semantics given so far can only describe the behaviors of a *closed* program $p$. What if $p$ calls a function outside of its domain? By definition, the execution goes wrong. As such, the procedural semantics alone is not compositional, and it is not enough to describe the behaviors of *open modules* (or *compilation units*).

In this section, we are going to make our procedural semantics compositional by extending it with a rule to handle external function calls, i.e. calls to functions that are not defined in the module.

This compositional semantics represents external function calls as events. We will later link two compilation units at the behavior level by replacing each external function call event with the behaviors of the callee (see Sec. 5).

The key idea of our compositional semantics is not to get stuck whenever a module calls an external function; instead, it produces a new form of event to record the external function call. This is consistent with the idea that events represent the interaction of a compilation unit with the outside environment: for an open module, external functions remain part of the outside environment until their implementation is provided by linking. These external call events are the minimal amount of syntax necessary to model external function calls at the level of behaviors.

For each external function call event, we record (1) the function name; (2) the memory state before the call, because the external call shall depend on the memory state under which it will be called; and (3) the memory state after function call. The external function may change the memory state arbitrarily, which the caller cannot control; but the behavior of the caller depends on how the callee changed the memory state.

For regular *input*, CompCert produces an ordinary event containing the value read from the environment. CompCert cannot predict the value, so it provides a behavior for every possible input. Which behavior appears at runtime will depend on the actual value read. More precisely, when a configuration requests an input, it must resort to nondeterminism and provide transition steps producing events for all possible values. Letting the event carry the value makes it possible to have different follow-up events or even termination status depending on the actual value read. For instance, in C, the command `scanf("%d", &i)` reads a value $j$ from the keyboard and stores it into the variable `i`. CompCert models it as follows: for every integer $j$, there is a transition producing the event $\texttt{IN}(j)$ asserting that $j$ is the value read. The following C code:

```
int main () {
  int i=0;
  scanf("%d", &i);
  printf("%d", (i%2));
  return 0;
}
```

will produce the set of behaviors [3]:

$$\{\texttt{IN}(j) :: \texttt{OUT}(j \bmod 2) :: \varepsilon \downarrow (0) : j \in [\mathsf{INT\_MIN}, \mathsf{INT\_MAX}]\}$$

[3] INT_MIN and INT_MAX are the least and the greatest values of type `int`.

We apply the same technique on external function calls. As the caller cannot predict how the callee will modify the memory state, the new memory state upon return of the function call is considered as an external input from the environment. This is why an external function call event stores both the new and old memory states.

Then, when an external function $f$ is called under some memory state $m_1$, for any possible memory state $m_2$ representing the memory state after returning from the external function, the compositional semantics will allow a transition (see rule EXTCALL in Def. 10 below) to produce the external function call event $\texttt{Extcall}(f, m_1, m_2)$. Consequently, the caller will be able to provide a behavior for each possible memory state $m_2$.

This leads us to extending the events and transition rules.

**Definition 8** (Extended events)**.** *Let $\mathfrak{L}$ be a language with function calls. We write $\mathbb{E}$ for the set of* extended events*, defined as follows:*

| | | | | |
|---|---|---|---|---|
| $\mathfrak{e}$ | $\in$ | $\mathbb{E}$ | | *Extended event* |
| | ::= | $e$ | $(e \in \mathbf{E})$ | *Regular event* |
| | \| | $\texttt{Extcall}(f, m_1, m_2)$ | $(f \in \mathbf{F},$ $m_1, m_2 \in \mathbf{MS})$ | *External function call* |

*In $\texttt{Extcall}(f, m_1, m_2)$, $m_1$ and $m_2$ are the memory states before and after the call, respectively. We write $\mathbb{B}$ for the set of behaviors on events $\mathbb{E}$, which we call* extended behaviors*.*

*If $F \subseteq \mathbf{F}$ is a set of function names, then we write $\mathbb{E}_F$ for the set of extended events where all external function call events $\texttt{Extcall}(f, m_1, m_2)$ have $f \in F$, and $\mathbb{B}_F$ the set of behaviors on such events.*

**Definition 9** (Module or compilation unit)**.** *Let $\mathfrak{L}$ be a language with function calls. A* module*, or* compilation unit*, is a partial function from function names to code*[4]*.*

**Definition 10** (Compositional semantics)**.** *The* compositional small-step semantics $\mathsf{Comp}[\mathfrak{L}, \mathfrak{u}]$ *of a compilation unit $\mathfrak{u}$ is the small-step semantics defined as follows:*

- *The set of events is $\mathbb{E}$.*
- *The set of configurations is $\mathbf{MS} \times \mathbf{LS} \times \mathbf{K}^*$ (as in the procedural small-step semantics).*
- *The transition relation $(\mathfrak{L}, \mathfrak{u}) \vdash_{comp} \cdot \to \cdot$ is defined as follows:*

$$\frac{(\mathfrak{L}, \mathfrak{u}) \vdash (m, l, \kappa) \xrightarrow{e} (m', l', \kappa') \quad e \in \mathbf{E}}{(\mathfrak{L}, \mathfrak{u}) \vdash_{comp} (m, l, \kappa) \xrightarrow{e} (m', l', \kappa')}$$

$$\frac{(\mathfrak{L}, \mathfrak{u}) \vdash (m, l, \kappa) \to (m', l', \kappa')}{(\mathfrak{L}, \mathfrak{u}) \vdash_{comp} (m, l, \kappa) \to (m', l', \kappa')}$$

$$\frac{\begin{array}{c}\mathsf{Kind}(l) = \texttt{Call}(f) \quad f \notin \mathrm{dom}(\mathfrak{u}) \\ \mathfrak{e} = \texttt{Extcall}(f, m, m') \\ l' = \mathsf{Restore}(m', \mathsf{Backup}(m, l))\end{array}}{(\mathfrak{L}, \mathfrak{u}) \vdash_{comp} (m, l, \kappa) \xrightarrow{\mathfrak{e}} (m', l', \kappa)} \quad (\text{EXTCALL})$$

- *As in the procedural semantics, the set of results is* **MS** *and the final configurations with result $m$ are the configurations $(m, l, \varepsilon)$ where $\mathsf{Kind}(l) = \texttt{Return}$.*

*The* compositional big-step semantics *of $\mathfrak{u}$ is the function $[\![\mathfrak{u}]\!]_{comp} : \mathrm{dom}(\mathfrak{u}) \longrightarrow \mathbf{MS} \longrightarrow \mathcal{P}(\mathbb{B}_{\mathbf{F} \setminus \mathrm{dom}(\mathfrak{u})})$ obtained from big-stepping the compositional small-step semantics.*

$$[\![\mathfrak{u}]\!]_{comp}(f)(m) = (\!|\mathsf{Comp}[\mathfrak{L}, \mathfrak{u}]|\!)(m, \mathsf{Init}(\mathfrak{u}(f), m), \varepsilon)$$

[4] Mathematically, modules and programs in $\mathfrak{L}$ are the same. But conceptually, a program is intended to be stand-alone, and is not expected to call functions that are not defined within itself, contrary to a compilation unit, which we view as an open module.

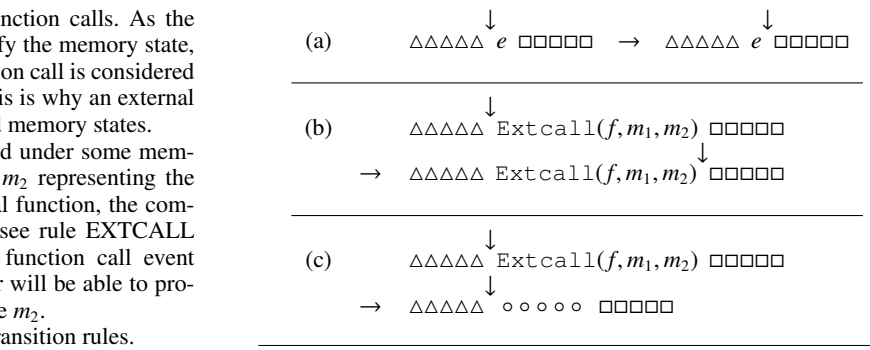


**Figure 1.** Three cases in behavior simulation: (a) regular event; (b) $f \notin \mathrm{dom}(\psi)$; (c) $\circ\circ\circ\circ\circ \in \psi(f)$.

## 5. Linking

In this section, we are going to define a *linking* operator $\bowtie$ between two partial functions from $\mathbf{F}$ to $(\mathbf{MS} \longrightarrow \mathcal{P}(\mathbb{B}))$. This linking operator will be defined directly at the level of the behaviors, independent of the underlying languages that the modules are written in.

Intuitively, each event corresponding to an external function call will be replaced with the behavior of the callee. However, plain straightforward substitution is not enough, as the behaviors of a compilation unit $\mathfrak{u}_1$ can involve external calls to functions defined in the other compilation unit $\mathfrak{u}_2$ that can again involve external calls to functions back in $\mathfrak{u}_1$. So, we have to *resolve* those formerly external calls that are now internal, namely the cross-calls between the two compilation units $\mathfrak{u}_1$ and $\mathfrak{u}_2$.

Let $F \subseteq \mathbf{F}$ be a set of function names. We are going to consider the functions in $\Psi(F) = F \longrightarrow (\mathbf{MS} \longrightarrow \mathcal{P}(\mathbb{B}))$ that describe the behaviors of functions of $F$. These functions may call some "external" functions which might still be in $F$. We call the elements of $\Psi(F)$ *open observations*, which we usually get by taking disjoint unions of multiple compilation unit semantics.

Let $\psi$ be such an open observation. We *resolve* the external calls (in $\psi$) to functions of $F$ by recursively supplying $\psi$ to do the substitution, yielding an observation $\mathcal{R}(\psi)$ in the set $\Phi(F) = F \longrightarrow (\mathbf{MS} \longrightarrow \mathcal{P}(\mathbb{B}_{\mathbf{F} \setminus F}))$ of *closed observations*, where there are no remaining external function call events to functions in $F$. We shall formally define $\mathcal{R}$ in definition 12.

Finally, if $\psi_1$ and $\psi_2$ are observations with disjoint domains, then we define the linking operator as $\psi_1 \bowtie \psi_2 = \mathcal{R}(\psi_1 \uplus \psi_2)$.

### 5.1 Internal call resolution by behavior simulation

Let $\psi \in \Psi(F)$ be an open observation. To resolve its internal function calls, we are going to define a semantics that will actually *simulate* the behaviors of $\psi$.

This resolution cancels out matching external call events by inlining each's behavior. We define this resolution by simulating the local behaviors of each module through a small-step semantics, treating each "external call" event through one "computation" step.

The simulation process is shown Fig. 1. In each case, $\downarrow$ can be seen as a cursor behind which lies the next event to be simulated. Each step $(\cdot \to \cdot)$ of the behavior simulation progresses based on the next event. All regular events are echoed as in (a), as well as all external function call events that correspond to functions not in $\psi$ (b). By contrast, each external function call event corresponding to a function defined in $\psi$ is replaced with the callee's events (c) where the cursor remains in the same spot ready to simulate the newly inserted events. Each step only performs one replacement at a time; the external function calls of the inlined behavior are not replaced yet until the cursor actually reaches them.

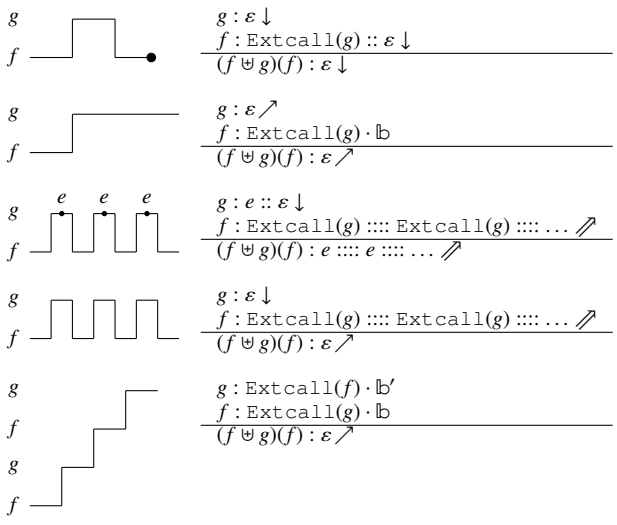


**Figure 2.** Examples of behaviors with external function calls between two compilation units, one defining $f$, another defining $g$, obtained by big-stepping the behavior simulation semantics. To simplify, we assume that $f$ and $g$ do not use any memory state.

Then, we obtain the resulting linked semantics by big-stepping this small-step semantics (see examples in Fig. 2).

Consider a function $f_0$ and a behavior $\sigma \bullet \texttt{Extcall}(f, m_1, m_2) \cdot \mathbb{b}$ being simulated. Assume that the prefix event sequence $\sigma$ has already been simulated so the `Extcall` is the first encounter of an external function call event where $f \in F$. Then $m_1$ is the memory state under which $f$ is to be called, and $m_2$ is the expected memory state upon return of $f$. Now, a behavior $\mathbb{b}_2$ is chosen in $\psi(f)(m_1)$, and is to be simulated, whereas the expected return memory state $m_2$ as well as the remaining behavior of the caller $\mathbb{b}$ to be simulated are pushed on top of a *continuation stack*. There are three cases:

- The simulation of this chosen behavior $\mathbb{b}_2$ terminates with the expected return memory state $m_2$. In this case, the remaining behavior $\mathbb{b}$ of the caller $f_0$ after the external call, popped from the continuation stack along with $m_2$, can be simulated.
- The simulation of this chosen behavior $\mathbb{b}_2$ goes wrong, diverges or reacts. In such cases, the simulation result of $\mathbb{b}_2$ takes over and never returns; the remaining behavior $\mathbb{b}$ of the caller after the external call event is discarded.
- The simulation of this chosen behavior $\mathbb{b}_2$ terminates, but with a return memory state that is not $m_2$. In this case, the remaining behavior of the caller is discarded, too, because it was relevant only in the case of termination with $m_2$. Actually, it means that the simulation of the caller behavior is *spurious*. This is because the set of behaviors of the caller $f_0$ has a behavior $\sigma \bullet \texttt{Extcall}(f, m_1, m_2') \cdot \mathbb{b}'$ for every $m_2'$, but most of the guesses are wrong. However, even though the simulation of the particular behavior does not make sense, the rule (EXTCALL) guarantees to have all possibilities covered, hence there will always be at least one behavior that is not spurious, it is OK to tag this irrelevant behavior as *spurious*. Formally, the simulation will not go wrong, but abruptly terminate with a special result `Spurious`. In the end, when big-stepping the small-step semantics, those spurious behaviors can be easily removed.

**Definition 11** (Behavior simulation)**.** *We define the* behavior simulation small-step semantics $\mathfrak{B}[\psi]$ *as follows:*

- *The set of events is* $\mathbb{E}$.
- *The set of configurations is defined as follows:*

$$\begin{array}{lcll} s & \in & \mathbf{S} & \\ & ::= & \texttt{Spurious} & \textit{Spurious state} \\ & | & (\mathbb{b}, \chi) \quad (\mathbb{b} \in \mathbb{B}, \ \chi \in (\mathbf{MS} \times \mathbb{B})^*) & \textit{Regular configuration} \end{array}$$

  *That is, either a special state for spurious executions, or a normal configuration with the current behavior* $\mathbb{b}$ *being simulated, paired with* $\chi$*, the stack of the remaining expected outcomes (if the current behavior simulations terminate) and the remaining behaviors to simulate.*
- *The transition relation* $\psi \vdash \cdot \longrightarrow \cdot$ *is defined as follows:*

$$\frac{\mathbb{e} = e \in \mathbf{E}}{\psi \vdash (\mathbb{e} \cdot \mathbb{b}, \chi) \xrightarrow{\mathbb{e}} (\mathbb{b}, \chi)}$$

$$\frac{f \notin \mathrm{dom}(\psi) \quad \mathbb{e} = \texttt{Extcall}(f, m_1, m_2)}{\psi \vdash (\mathbb{e} \cdot \mathbb{b}, \chi) \xrightarrow{\mathbb{e}} (\mathbb{b}, \chi)}$$

$$\frac{\mathbb{b}' \in \psi(f)(m_1)}{\psi \vdash (\texttt{Extcall}(f, m_1, m_2) \cdot \mathbb{b}, \chi) \rightarrow (\mathbb{b}', ((m_2, \mathbb{b}) :: \chi))}$$

$$\frac{}{\psi \vdash (\varepsilon\downarrow(m), ((m, \mathbb{b}) :: \chi)) \rightarrow (\mathbb{b}, \chi)} \quad \text{(RETURN)}$$

$$\frac{}{\psi \vdash (\varepsilon\nearrow, \chi) \rightarrow (\varepsilon\nearrow, \chi)}$$

$$\frac{m' \neq m}{\psi \vdash (\varepsilon\downarrow(m'), ((m, \mathbb{b}) :: \chi)) \rightarrow \texttt{Spurious}} \quad \text{(RETURN-SPURIOUS)}$$

- *The set of results is defined as follows:*

$$\begin{array}{lcll} r & \in & \mathbf{R} & \\ & ::= & \texttt{Spurious} & \textit{Spurious behavior} \\ & | & m \quad (m \in \mathbf{MS}) & \textit{Regular termination} \end{array}$$

- *The behavior sequence* $((\varepsilon\downarrow(m)), \varepsilon)$ *is the only final state with result* $m \in \mathbf{MS}$*.* `Spurious` *is the only final state with result* `Spurious`*.*

**Definition 12** (Resolution)**.** *Let* $\mathbb{B}^{\texttt{Spurious}} = \{\sigma \downarrow \texttt{Spurious} : \sigma \in \mathbb{E}^*\}$ *be the set of all spurious behaviors.*

*Then, the* resolution *of an open observation* $\psi \in \Psi(F)$ *is the closed observation* $\mathcal{R}(\psi) \in \Phi(F)$ *defined using the big-step semantics of the behavior simulation small-step semantics, excluding spurious behaviors:*

$$\mathcal{R}(\psi)(f)(m) = \bigcup_{\mathbb{b} \in \psi(f)(m)} (\!|\mathfrak{B}[\psi]|\!)(\mathbb{b}, \varepsilon) \backslash \mathbb{B}^{\texttt{Spurious}}$$

### 5.2 Semantic linking

Thanks to the resolution operator, we can simply define the linking of two observations:

**Definition 13** (Linking)**.** *Let* $\psi_1, \psi_2$ *be two observations with disjoint domains. Then, their linking* $\psi_1 \bowtie \psi_2$ *is defined as:*

$$\psi_1 \bowtie \psi_2 = \mathcal{R}(\psi_1 \uplus \psi_2)$$

With the definition of linking at the level of behaviors, we can show that the compositional semantics of a compilation unit is indeed compositional. In other words, in the special case where the two modules are in the same language, linking their compositional semantics at the level of their behaviors exactly corresponds to the compositional semantics of the syntactic concatenation of the two compilation units, which conforms to the intuition of linking:

**Theorem 1.** *If* $\mathfrak{u}_0, \mathfrak{u}_1$ *are two compilation units with disjoint domains in the same language with function calls, then:*

$$[\![\mathfrak{u}_0 \uplus \mathfrak{u}_1]\!]_{comp} = [\![\mathfrak{u}_0]\!]_{comp} \bowtie [\![\mathfrak{u}_1]\!]_{comp}$$

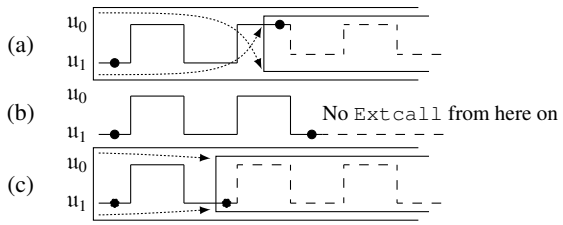


**Figure 3.** Illustrations of the three cases in the $\subseteq$ branch of the proof of theorem 1: (a) one of the call to $\mathfrak{u}_0$ never returns; (b) all external calls return with finitely many of them; (c) all external calls return with infinitely many of them. Dotted arrows denote coinduction hypothesis.

*Proof (in Coq).* • $\supseteq$: We introduce a simulation diagram: an execution step in $[\![\mathfrak{u}_0]\!]_{\mathsf{comp}} \bowtie [\![\mathfrak{u}_1]\!]_{\mathsf{comp}}$ matches at least one execution step in $[\![\mathfrak{u}_0 \uplus \mathfrak{u}_1]\!]_{\mathsf{comp}}$. In this simulation diagram, we maintain an invariant between the configuration state $(\mathbb{b}, \chi)$ in $[\![\mathfrak{u}_0]\!]_{\mathsf{comp}} \bowtie [\![\mathfrak{u}_1]\!]_{\mathsf{comp}}$ and the configuration state $(m, l, \kappa)$ in $[\![\mathfrak{u}_0 \uplus \mathfrak{u}_1]\!]_{\mathsf{comp}}$ such that $\kappa$ can be decomposed in $\kappa' +\!\!+ \kappa''$ with $\mathbb{b}$ being a valid behavior in $[\![\mathfrak{u}_i]\!]_{\mathsf{comp}}$ from $(m, l, \kappa')$ and $\kappa''$ matching the stack $\chi$.

• $\subseteq$: the result for terminating and stuck behaviors is proven by induction on the length of the execution. On the other hand, diverging and reacting behaviors are dealt with in the following way. Starting from such a behavior $\mathbb{b}$ in $[\![\mathfrak{u}_0 \uplus \mathfrak{u}_1]\!]_{\mathsf{comp}}$, we first isolate an infinite step sequence corresponding to $\mathbb{b}$ by definition. Given $i \in \{0, 1\}$, we build a behavior $\mathbb{b}'$ in $[\![\mathfrak{u}_i]\!]_{\mathsf{comp}}$ by replacing the calls to functions in $\mathfrak{u}_{1-i}$ with external calls, and we prove that simulating $\mathbb{b}'$ in $\mathfrak{B}[[\![\mathfrak{u}_0]\!]_{\mathsf{comp}} \uplus [\![\mathfrak{u}_1]\!]_{\mathsf{comp}}]$ yields $\mathbb{b}$, i.e. $\mathbb{b} \in (\!|\mathfrak{B}[[\![\mathfrak{u}_0]\!]_{\mathsf{comp}} \uplus [\![\mathfrak{u}_1]\!]_{\mathsf{comp}}]|\!)(\mathbb{b}', \varepsilon)$. There are three cases (each of which is illustrated in Fig. 3):

(a) There is a call to a function in $\mathfrak{u}_{1-i}$ that never terminates. So, before the first such external call, we can build the finite prefix of a behavior in $[\![\mathfrak{u}_i]\!]_{\mathsf{comp}}$, and deal with this external function call by coinduction replacing $i$ with $1 - i$.

(b) All calls to functions in $\mathfrak{u}_{1-i}$ terminate and there are finitely many of them. So, until the last such external function call, we can build the finite prefix of a behavior in $[\![\mathfrak{u}_i]\!]_{\mathsf{comp}}$. Then, we prove that the remaining behavior of $[\![\mathfrak{u}_0 \uplus \mathfrak{u}_1]\!]_{\mathsf{comp}}$ that calls no functions of $\mathfrak{u}_{1-i}$ is actually a behavior in $[\![\mathfrak{u}_i]\!]_{\mathsf{comp}}$.

(c) All calls to functions in $\mathfrak{u}_{1-i}$ terminate but there are infinitely many of them. So, we have to build a reacting behavior in $[\![\mathfrak{u}_i]\!]_{\mathsf{comp}}$ with infinitely many external function calls to $\mathfrak{u}_{1-i}$, each one replacing each call to a function in $\mathfrak{u}_{1-i}$.

□

We could have used *behavior trees* to model external function calls. Behavior trees are well-known to be used in denotational semantics to model input. They would have turned events for external function calls $\texttt{Extcall}(f, m_1, m_2)$ into branching nodes, with each branch labeled with the memory state $m_2$. Using behavior trees instead of plain behaviors would have helped remove spurious behaviors, as the two rules (RETURN) and (RETURN-SPURIOUS) would have been replaced by a single rule actually choosing the right branch in the behavior tree. However, this would require adopting behavior trees as the semantic object for the compositional semantics. Then, the process of making the procedural semantics into a compositional semantics would bring deep changes to the procedural small-step semantics, and the current compiler correctness proofs of CompCert based on simulation diagrams over those small-step semantics would require deep changes as well.

We believe that our current per-behavior setting, where behaviors are represented as first-order objects, shall require less intrusive changes in the current CompCert proofs. From the compiler's point of view, the external function call events introduced by our semantics need not be treated differently from ordinary events.

The relationship between the compositional semantics and the procedural semantics of a module viewed as a whole program is rather obvious: it suffices to link the compilation unit with an observation that makes every external function call stuck.

**Lemma 2** (Compositional and procedural semantics)**.** *Let* $\mathfrak{u}$ *be a compilation unit in some language with function calls. Define* $\psi_\lightning$ *the constant stuck observation:*

$$\forall f \notin \mathrm{dom}(\mathfrak{u}), \forall m : \psi_\lightning(f)(m) = \{\varepsilon\lightning\}$$

*Then,* $\forall f \in \mathrm{dom}(\mathfrak{u}) : [\![\mathfrak{u}]\!](f) = ([\![\mathfrak{u}]\!]_{comp} \bowtie \psi_\lightning)(f)$.

## 6. Refinement and compiler correctness

The term "refinement" in program development dates back to the early 70s proposed by Dijkstra [6] and Wirth [22]. It quickly grows in various fields [16]. Refinement also plays a heavy role in compiler verification as shown in CompCert [13] and Müller-Olm [17].

In this work, we use refinement to define and prove correctness of separate compilation. We first state the necessary conditions for a relation to be a refinement relation. Then we show how our refinement framework applies to compiler correctness. Finally, we show that the behavior refinement relation defined in CompCert extends well to the setting of our compositional semantics.

### 6.1 Refinement relations

Instead of defining on pairs of programs or specifications, we define our refinement relations on sets of extended behaviors. One reason for this choice is to support refinement between multiple languages and program logics. Another reason is to better handle interactions between refinement relations and the linking operator — or, more generally, between refinement relations and the resolution operator, which we will discuss at the end of this subsection.

To generalize refinement to the compositional semantics instead of sticking to the procedural semantics of a whole program, we define refinement relations on sets of extended behaviors instead of plain behaviors.

Let $\sqsubseteq$ be a binary relation on $\mathcal{P}(\mathbb{B})$. Then, we lift it to observations in a straightforward way: we define $\psi_1 \sqsubseteq \psi_2$ if, and only if, $\mathrm{dom}(\psi_1) = \mathrm{dom}(\psi_2)$ and:

$$\forall f \in \mathrm{dom}(\psi_1), \forall m : \psi_1(f)(m) \sqsubseteq \psi_2(f)(m)$$

**Definition 14** (Refinement relations)**.** *A binary relation* $\sqsubseteq$ *on* $\mathcal{P}(\mathbb{B})$ *is a refinement on observable behaviors if all these hold:*

- *reflexivity:* $\forall \mathbb{B}_0 \in \mathcal{P}(\mathbb{B}), \mathbb{B}_0 \sqsubseteq \mathbb{B}_0$
- *transitivity:* $\forall \mathbb{B}_1, \mathbb{B}_2, \mathbb{B}_3 \in \mathcal{P}(\mathbb{B})$ :

$$\mathbb{B}_1 \sqsubseteq \mathbb{B}_2 \wedge \mathbb{B}_2 \sqsubseteq \mathbb{B}_3 \Longrightarrow \mathbb{B}_1 \sqsubseteq \mathbb{B}_3$$

- *congruence: for any observations* $\psi_1, \psi_2$ *such that* $\psi_1$ *is never empty (*$\forall f \in \mathrm{dom}(\psi_1), \forall m, \psi_1(f)(m) \neq \varnothing$*):*

$$\psi_1 \sqsubseteq \psi_2 \Longrightarrow \mathcal{R}(\psi_1) \sqsubseteq \mathcal{R}(\psi_2)$$

On top of a preorder, we add the congruence property, thanks to which we can easily show that refinement is compositional:

**Theorem 2** (Compositionality of refinement)**.** *Let* $\psi_1, \psi_2$ *two observations such that* $\psi_1 \sqsubseteq \psi_2$ *and* $\psi_1$ *is never empty. Then, for any never-empty observation* $\psi$ *with a domain disjoint from* $\psi_1$*, we have* $\psi \bowtie \psi_1 \sqsubseteq \psi \bowtie \psi_2$.

We will see in Sec. 6.4 that the CompCert improvement relation is actually a refinement relation meeting all those requirements.

### 6.2 Compositional program verification

Refinement is expressed at the level of extended behaviors. So, we can consider that a *specification* is an open observation, so that a compilation unit $\mathfrak{u}_1$ in some language with function calls can be said to make a specification $\psi_1$ hold if $[\![\mathfrak{u}_1]\!]_{\mathsf{comp}} \sqsubseteq \psi_1$. (The specification $\psi_1$ can still contain external function call events to functions outside of $\mathrm{dom}(\mathfrak{u}_1)$.)

Thanks to refinement compositionality, and to the fact that the linking operator is defined at the semantic level of extended behaviors, this program verification scheme is compositional and suitable for open modules. Indeed, a compilation unit $\mathfrak{u}_2$ with a domain disjoint from $\psi_1$ can be proven to make some specification $\psi$ hold under assumptions that it is linked with an unknown library verifying $\psi_1$ independently of the actual implementation of the library: we can directly link the compilation unit $\mathfrak{u}_2$ with the specification $\psi_1$ and prove that $\psi_1 \bowtie [\![\mathfrak{u}_2]\!]_{\mathsf{comp}} \sqsubseteq \psi$. Then, if $[\![\mathfrak{u}_1]\!]_{\mathsf{comp}} \sqsubseteq \psi_1$, refinement compositionality gives us the refinement proof on the linked program: $[\![\mathfrak{u}_1]\!]_{\mathsf{comp}} \bowtie [\![\mathfrak{u}_2]\!]_{\mathsf{comp}} \sqsubseteq \psi$; finally, in the special case where $\mathfrak{u}_1$ and $\mathfrak{u}_2$ are written in the same language, we have $[\![\mathfrak{u}_1 \uplus \mathfrak{u}_2]\!]_{\mathsf{comp}} \sqsubseteq \psi$.

### 6.3 Verified separate compilation

In this work, we follow the common notion of compiler correctness that a compiler is correct if all possible behaviors of the target program are valid behaviors of the source program. Since language specifications often leave some decisions to compilers for flexibility, a compiler is allowed to remove behaviors. In other words, compilation is a refinement step.

Our compiler correctness definition is fairly standard except for the abstract refinement relation instead of a plain subset relation. As it uses a refinement relation on extended behaviors, compiler correctness generalizes to compiling open modules by considering their compositional semantics.

**Definition 15** (Compiler (optimizer) correctness)**.** *Let $\mathfrak{L}, \mathfrak{L}'$ be two languages with function calls.*

*Under a refinement relation $\sqsubseteq$, a compiler $C$ from $\mathfrak{L}$ to $\mathfrak{L}'$ is said to be correct if and only if, for any compilation unit $\mathfrak{u}$, $[\![C(\mathfrak{u})]\!]_{comp} \sqsubseteq [\![\mathfrak{u}]\!]_{comp}$.*

**Theorem 3.** *Under a refinement relation, multiple correct compilers are compatible with separate compilation. If $\mathfrak{u}_1, \ldots, \mathfrak{u}_n$ are compilation units with disjoint domains and $C_1, \ldots, C_n$ are all correct compilers, then: $[\![C_1(\mathfrak{u}_1) \uplus \cdots \uplus C_n(\mathfrak{u}_n)]\!] \sqsubseteq [\![\mathfrak{u}_1 \uplus \cdots \uplus \mathfrak{u}_n]\!]$*

*Proof.* By definition of compiler correctness, $\forall i, [\![C_i(\mathfrak{u}_i)]\!]_{\mathsf{comp}} \sqsubseteq [\![\mathfrak{u}_i]\!]_{\mathsf{comp}}$. By transitivity and multiple applications of refinement compositionality (Theorem 2), we obtain

$$[\![C_1(\mathfrak{u}_1)]\!]_{\mathsf{comp}} \bowtie \ldots \bowtie [\![C_n(\mathfrak{u}_n)]\!]_{\mathsf{comp}} \sqsubseteq [\![\mathfrak{u}_1]\!]_{\mathsf{comp}} \bowtie \ldots \bowtie [\![\mathfrak{u}_n]\!]_{\mathsf{comp}}$$

which leads to $[\![C_1(\mathfrak{u}_1) \uplus \cdots \uplus C_n(\mathfrak{u}_n)]\!]_{\mathsf{comp}} \sqsubseteq [\![\mathfrak{u}_1 \uplus \cdots \uplus \mathfrak{u}_n]\!]_{\mathsf{comp}}$ because of Theorem 1 (linking in the same language). Finally, to go from the compositional to the procedural semantics, refinement compositionality with $\psi_{\lightning}$ and Lemma 2 give the result. □

The theorem tells us that we can link several object files which are compiled independently with potentially different compilers. As long as all the compilers are correct, the linked executable will behave as an instance of the program linked at the source level.

With a single correct compiler $C$, Theorem 3 ensures the correctness of separate compilation even though we may not have $C(\mathfrak{u}_1) \uplus \cdots \uplus C(\mathfrak{u}_n) = C(\mathfrak{u}_1 \uplus \cdots \uplus \mathfrak{u}_n)$ (e.g. if $C$ performs some function inlining).

### 6.4 Example: the CompCert refinement relation

When developing a compiler, it is usually hard or even impossible to retain one kind of behavior – the stuck behaviors. Imagine a C program that takes the address of a local variable, adds a constant to it, and then uses the result as the address to write to. If the arithmetic operation brings the address out of bound, the C semantics will get stuck. While in the target assembly code, the program is likely to continue running and crash at a much later point, or even keep going normally as the place the program writes to might be an unused stack space.

In CompCert [12], all behaviors with the event sequence before crashing as a prefix are considered "improvements" of the crashing behavior. The refinement relation it uses, initially proposed by Dockins [7] and integrated into CompCert, incorporates improvements and is an extension of a subset relation. In this section, we extend it to extended behaviors with external function call events.

**Definition 16** (Behavior improvement)**.** *Let $\mathbb{b}_1, \mathbb{b}_2$ be two extended behaviors. $\mathbb{b}_1$ improves $\mathbb{b}_2$ ($\mathbb{b}_1 \sqsubseteq \mathbb{b}_2$) if and only if:*

- *either $\mathbb{b}_1 = \mathbb{b}_2$, or*
- *$\mathbb{b}_2$ is a "stuck prefix" of $\mathbb{b}_1$: there exists an event sequence $\sigma$ and a behavior $\mathbb{b}$ such that $\mathbb{b}_2 = \sigma\lightning$ and $\mathbb{b}_1 = \sigma \bullet \mathbb{b}$.*

**Definition 17** (CompCert improvement relation)**.** *Let $\mathbb{B}_1, \mathbb{B}_2$ be two sets of extended behaviors. $\mathbb{B}_1$ improves $\mathbb{B}_2$ ($\mathbb{B}_1 \sqsubseteq \mathbb{B}_2$) if, and only if $\forall \mathbb{b}_1 \in \mathbb{B}_1, \exists \mathbb{b}_2 \in \mathbb{B}_2 : \mathbb{b}_1 \sqsubseteq \mathbb{b}_2$.*

**Theorem 4.** *The CompCert improvement relation is a refinement relation.*

*Proof (in Coq).* Congruence is proven by a lock-step backwards simulation, where the invariant between two configurations of the semantics uses behavior improvement for the behavior being simulated as well as every frame of the continuation stack. □

This theorem shows that the CompCert improvement relation defined on behaviors extends well to extended behaviors and verified separate compilation. Consequently, a correct compiler can compile an open module as if it were a whole program, by considering an external call event in no different way than a regular event. By the way, it also shows that a correct compiler necessarily preserves external function calls: in no way can it optimize them away before linking with an actual implementation for them. This is understandable because a compiler processing an open module has no hypotheses about external functions.

### 6.5 Coq implementation

Our Coq implementation provides the following enhancements, which we did not mention for the sake of presentation.

Functions can be passed arguments, and they can return a value. Then, the arguments are additional parameters to the semantics of a module, and they appear in the external function call events as well as the return values. Similar to the resulting memory state upon return of an external function call, the caller has to provide a behavior for each possible return value as well: given an external function $f$ called with arguments $arg$ and the memory state $m_1$, the external function call rule (EXTCALL in Def. 10) of the compositional semantics produces an event $\texttt{Extcall}(f, arg, m_1, ret, m_2)$ for any result $ret$ and any memory state $m_2$.

Throughout the execution of a language with function calls, we added the ability of maintaining some *invariant* on the memory state. We equip the set of memory states with some preorder $\preceq$, such that, whenever an internal step is performed from a memory state $m$, the new memory state $m'$ is such that $m \preceq m'$. Consequently, the semantics of a compilation unit provides no behavior for those external function calls that do not respect $\preceq$: in the compositional semantics, the rule for external function calls (EXTCALL in Def. 10) producing an external call event $\texttt{Extcall}(f, arg, m_1, ret, m_2)$ requires the additional premise $m_1 \preceq m_2$. This enhancement is important

for CompCert, where the memory model requires that the memory evolve monotonically to prevent a deallocated memory block from being reused. The proofs of compilation passes in CompCert make critical use of this assumption.

In a language with function calls, the functions Backup and Restore which respectively save the local state into a continuation stack frame and retrieve a new one from such a frame, can change the memory state: instead of only returning a frame or local state, they return a new memory state as well. We make those functions compatible with the preorder $\preceq$ over memory states. This enhancement allows us to model the allocation and deallocation of a concrete stack frame in the memory upon function call and return.

We also provide a Coq implementation to instantiate our framework with the CompCert common subexpression elimination pass to turn it from whole-program compilation to separate compilation.

This pass is carried over CompCert RTL ("register transfer language") as both source and target languages. It is a 3-address language with infinitely many per-function-call pseudo-registers. The body of a function is a control-flow graph.

The common subexpression elimination actually replaces nodes of the control-flow graph with no-ops, if those nodes are taking part to expressions that were already computed before. This pass actually does not alter function calls and does not modify the memory between source and target programs.

We took a subset of RTL eliminating floating-point operations (due to typing constraints). Then, we added our external function call event to the CompCert so-called "external functions" (namely primitives such as volatile load and store, memory copy, or I/O, some of which generate events) to enable their support by RTL. Then, we rewrote RTL into the setting of our framework and proved that the corresponding compositional semantics and the CompCert RTL language with those new events produce the same big-step semantics. Thus, there were no changes to the proof of the compilation pass (except the removal of floating-point operations) and the correctness of separate compilation were stated directly in terms of the original RTL semantics and proved using our framework.

# 7. Languages with different memory state models

Our new approach is close to the way how CompCert [13] handles I/O events. Actually, we generalize it to arbitrary external function calls, and we give the formal argument why this approach is correct by enabling those external functions to be implemented and their behaviors inlined. This means that the compiler correctness techniques used for CompCert and restricted to whole programs can be easily applied to open modules.

The main difference introduced by considering the behaviors of open modules is that now part of the memory state becomes observable. There still remains a problem: a compilation pass can alter the observable memory state.

But alterations can deeply involve the structure of the memory state so that the relation between the memory states of the source programs and the compiled ones can itself change during execution. Such relations are called *Kripke worlds* [2, 11] in the setting of Kripke logical relations. But it becomes necessary to define the refinement relation as a "binary" simulation diagram deprecating the notion of the "unary" semantics.

In this section, we show that such Kripke logical relations are not necessary to deal with critical memory-changing passes of CompCert. To this purpose, we introduce a lightweight infrastructure to deal with memory-changing relations, $\alpha$-refinement, that can directly cope with our unary semantics for open modules with traces of external function call events.

## 7.1 $\alpha$-refinement

In practice, a separate compiler does make some assumptions on the behaviors of external functions. If these assumptions are also preserved as an invariant by the execution of functions defined in $\mathfrak{u}$, the compilation of $\mathfrak{u}$ can take advantage of this invariant.

Consider a module $\mathfrak{u}$ written in a procedural language $\mathfrak{L}$. Let $MS$ be the set of memory states of $\mathfrak{L}$. Let $I \subseteq MS$ be an *invariant* in $\mathfrak{L}$, i.e. such that for any local transition $(m, l) \to (m', l')$, if $m \in I$ then $m' \in I$. Then we can restrict the set of memory states of $\mathfrak{L}$ to $I$, yielding a procedural language $\mathfrak{L}|_I$ such that the corresponding *compositional* semantics mandates all external function calls to return with memory states also satisfying the invariant. In other words, for any external function call event $\texttt{Extcall}(f, m_1, m_2)$ produced by the compositional semantics of $\mathfrak{L}|_I$, we always have $m_1, m_2 \in I$.

Now consider a target procedural language $\mathfrak{L}'$ having an invariant $I'$. Let $C$ be a compiler from $\mathfrak{L}$ to $\mathfrak{L}'$. Then, we say that $C(\mathfrak{u})$ *$\alpha$-refines* $\mathfrak{u}$ ($C(\mathfrak{u}) \sqsubseteq_\alpha \mathfrak{u}$) if, and only if there exists a bijection $\alpha$ between $I$ and $I'$ such that $[\![C(\mathfrak{u})]\!]_{\mathsf{comp}}|_{I'} \sqsubseteq \alpha([\![\mathfrak{u}]\!]_{\mathsf{comp}}|_I)$.

In practice, it means that the separate compiler $C$ is correct when the modules are linked with other modules also satisfying the same invariants ($I$ in the source, $I'$ in the target). Indeed, in the case when such a bijection $\alpha$ exists, then we can define the procedural language $\alpha(\mathfrak{L}|_I)$ isomorphic to $\mathfrak{L}|_I$ where the set of memory states is $\alpha(I) = I'$, and then we can use the usual non-memory-changing refinement relation between $\alpha(\mathfrak{L}|_I)$ and $\mathfrak{L}'|_{I'}$. Then, separate compilation is correct provided that, when building the whole program by linking with a module containing a `main` entry point, the initial memory state passed to `main` also satisfies the invariant ($I$ in the source, $I'$ in the target).

Then, Theorem 3 can be rephrased as follows: if $\mathfrak{u}_1, \ldots, \mathfrak{u}_n$ are compilation units in languages $\mathfrak{L}_1, \ldots, \mathfrak{L}_n$ with disjoint domains and $C_1, \ldots, C_n$ are all compilers to the same target language $\mathfrak{L}'$ such that, for each $i$, $C_i$ is correct with respect to an $\alpha_i$-refinement, then:

$$[\![C_1(\mathfrak{u}_1) \uplus \cdots \uplus C_n(\mathfrak{u}_n)]\!] \sqsubseteq \alpha_1([\![\mathfrak{u}_1]\!]) \bowtie \ldots \bowtie \alpha_n([\![\mathfrak{u}_n]\!])$$

In the rest of this section, we show how to systematically turn CompCert-style memory injection into $\alpha$-bijection by using a critical memory-changing pass of CompCert as an example. The same technique can also be used to support translation of calling conventions (e.g., mapping local variables or temporaries in the source into stack entries in the target).

## 7.2 Case study: memory injection for local variable layout

One of the most critical memory-changing compilation phases in CompCert is the phase that lays out local variables into a stack frame. Indeed, CompCert does not represent memory as a unique byte array, but as a collection of byte arrays called *memory blocks*. The purpose of this memory model is to allow pointer arithmetic only within the same block. In this setting, CompCert defines the semantics of a subset of C by allocating one block for each local variable, so that the following code example indeed gets stuck (has no valid semantics, which corresponds to *undefined behavior* according to the C standard):

```
void f (void)
{ int a[2] = {18, 42}, b[2] = {1729, 6};
  register int *pa = &a[2], *pb = &b[0];
  *pa = 3; /* undefined behavior,
              NOT equivalent to *pb = 3 */ }
```

In this example, upon function entry, CompCert allocates two different memory blocks, one (say with identifier 2) of size 8 for `a` and one (say with identifier 3) of size 4 for `b`. Then, the pointer `pa` contains an address which is, in CompCert, not a plain integer, but a *pair* $\mathsf{Vptr}(b, o)$ of the block identifier $b$ and the byte offset $o$ within

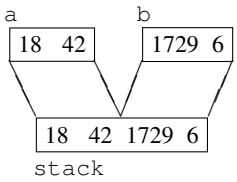


**Figure 4.** Injecting two arrays into the stack

this block. So, the value of `pa` is actually $\mathsf{Vptr}(2, 8)$ whereas `pb` is $\mathsf{Vptr}(3, 0)$. So, the two pointers are not equal, and in fact, `pa` is not a valid pointer to store to, because the size of the block identifier corresponding to `a` is 8. In other words, the boundary of one block is in no way related to other blocks. This instrumented semantics can help in tracking out-of-bounds array accesses in a C program (for instance using the reference interpreter included in CompCert to "animate" the formal semantics of CompCert C).

But in practice, this C code is actually compiled by CompCert to an intermediate language called Cminor, which performs pointer arithmetic and memory operations for stack-allocated variables of a given function call in one single memory block, called the stack frame. The compiled Cminor code looks like the following C code:

```
void f (void)
{ char* stk[16];
  *(int*)(&stk[0]) = 18;   *(int*)(&stk[4]) = 42;
  *(int*)(&stk[8]) = 1729; *(int*)(&stk[12]) = 6;
  { register int* pa = (int*)(&stk[8]);
    register int* pb = (int*)(&stk[8]);
    *pa = 3;                                    }}
```

The proof of the Cminor code generation, from the Csharpminor [5] intermediate language still having one memory block for each local variable, is based on a memory transformation called *memory injection* [14]. An injection is a partial function $\iota : \mathit{BlockID} \to (\mathit{BlockID} \times \mathbb{Z})$ mapping a source memory block to an offset within a target memory block. In our example, the target memory block allocates a stack frame (say with block-id 2) of size 16 bytes; the source memory block for `a` is mapped to offset 0 within this stack block, and `b` is mapped to offset 8: $\iota(2) = (2, 0)$ and $\iota(3) = (2, 8)$.

### 7.3 Issues

Although the CompCert memory injection is the most critical memory transformation used in CompCert and makes formal proofs of whole-program compilation fairly understandable (but by no means straightforward), it has several issues that make it difficult to turn those proofs into separate compilation (in the sense that it is difficult to turn the memory injection into a bijective memory transformation amenable to $\alpha$-refinement).

***Granularity of preservation by memory operations*** In the current correctness proof of the Csharpminor-to-Cminor pass, the memory injection is kept as an invariant, but the preservation properties make the memory injection hold even during the allocation of memory blocks corresponding to the source local variables. More precisely, assume that `main` is called from source memory $m_0$ related to target memory $m'_0$ by a memory injection $\iota_0$. Then:

1. First, the stack frame block $b'$ is created in the target memory which becomes $m'_1$. Memory injection $\iota_0$ still holds between $m_0$ and $m'_1$.
2. Then, the memory block for the local variable `a`, say $b_2$, is created in the source memory which becomes $m_2$. Memory injection between $m_2$ and $m'_1$ becomes $\iota_2 = \iota_0 \uplus (b_2 \mapsto (b', 0))$.
3. Then, the memory block for the local variable `b`, say $b_3$ is created in the source memory which becomes $m_3$, injected into $m'_1$ through $\iota_3 = \iota_2 \uplus (b_3 \mapsto (b', 8))$

The current memory injection invariant is too fine-grained because it also holds in the middle of allocating the memory blocks for the source local variables. It actually means that the target memory $m'_1$ is related to any source memory that can be obtained in the middle of the allocation of such source blocks, which prevents the injectivity of the memory transformation. Conversely, the allocation of the target stack frame block is performed without changing the source memory, so that the memory injection is not even functional.

To remedy this problem, we make the preservation lemma for memory injection more coarse-grained: instead of specifying a per-allocation preservation property, we specify an all-in-one preservation property to reestablish injection only after all the blocks corresponding to source local variables are allocated.

***Dynamic memory changes*** The proofs of compilation passes involving memory injections build the block mapping on the fly during the execution of the program: whenever a block is allocated, the mapping is modified accordingly. But the mapping is not yet known for those source memory blocks that are not allocated yet, e.g. in future function calls, or heap allocations (malloc and free library functions). It means that the mapping dynamically changes during the execution of a program. This is why Kripke logical relations are used to handle memory-changing compilation passes.

To solve those issues, we propose to define a stronger notion of memory injection in two steps. First, the block mapping is computed from the source memory using additional information contained in block tags. Then, the target memory is computed from both the source memory and the computed block mapping.

### 7.4 Our approach

In fact, the memory transformation for the Csharpminor-to-Cminor is actually systematic and can be defined directly depending on the shape of the memory itself rather than specified by an invariant preserved by memory operations such as allocating a new block. To this purpose, we need to add more information into the memory under the form of *tags* attached to each memory block. Such information is provided by the language semantics when allocating a new memory block, and no longer changes during the execution of the program. It plays little active role in the execution of the program, as it is only used during the compilation proof.

***Block identifiers*** To make proofs simpler, we modify the semantics of Csharpminor and Cminor to keep the block identifiers synchronized so that as many blocks are "allocated" in the source as in the target. In the source, an empty block (within which no operation or pointer arithmetic is valid) is first allocated, then the blocks for local variables are allocated; whereas in the target, the stack frame block is first allocated with its size, then many empty blocks are allocated, one for each variable.

This has no incidence on performance: such empty blocks can be considered as *logical information*, which correspond to no memory in practice. They are not even reachable in the program.

***Tags*** A block has a tag of one of the following forms:

$t \in T$ ::= $\mathsf{Heap}$
  global variable or free store
| $\mathsf{Stack}(\mathsf{Main}(f, sz))$
  Stack frame for function $f$ of size $sz$ bytes
| $\mathsf{Stack}(\mathsf{Var}(f, id, b, sz, of))$
  Local variable $id$ in $f$ of size $sz$ injected into $b$ at offset $of$

[5] This language is a C-like language only keeping simple types: unions and structures are removed and compiled to casts

Information defined in the tags is provided either by the semantics of Csharpminor (e.g. the identifier $b'$ of the corresponding Main block in the tags of Var blocks) or by a previous compilation phase (e.g. offsets) within Csharpminor without changing the actual contents of memory blocks.

***Specification of injection*** We can now replace CompCert memory injection with a stronger injection $\mathsf{INJ}(\iota, m, m')$ between a source memory $m$ and a target memory $m'$ axiomatized as follows:

- The empty memory injects into itself with $\iota = \varnothing$.
- If $\mathsf{INJ}(\iota, m, m')$, then $\mathsf{INJ}(\iota \uplus (b \mapsto (b, 0), m \uplus \{b\}, m' \uplus \{b\}))$ for any allocation of a new block $b$ with tag Heap
- If $\mathsf{INJ}(\iota, m, m')$, then, if the source allocates one empty block $b$ with tag $\mathsf{Stack}(\mathsf{Main}(f, sz))$ and several blocks corresponding to the local variables of $f$ with tags $\mathsf{Stack}(\mathsf{Var}(f, id_i, sz_i, of_i))$ so that $[of_i, of_i + sz_i)$ are a partition of $[0, sz)$, then the target memory allocating one block $b$ of size $sz$ and as many empty blocks is related to the resulting source memory by INJ with $(\iota \uplus \{(b + i \mapsto (b, of_i)) : 1 \leq i \leq n\})$.
- Load, store and free operations are preserved with respect to $\iota$
- If $\mathsf{INJ}(\iota, m, m_1)$ and $\mathsf{INJ}(\iota, m, m_2)$, then $m_1 = m_2$
- If $\mathsf{INJ}(\iota_1, m_1, m)$ and $\mathsf{INJ}(\iota_2, m_2, m)$, then $(\iota_1, m_1) = (\iota_2, m_2)$
- The block tags of the source and target memories are the same

Then, the memory transformation is defined as the partial injective function $\alpha(m) = \{m' : \exists \iota, \mathsf{INJ}(\iota, m, m')\}$. Then, we change the forward simulation proof of the CompCert Csharpminor-to-Cminor pass by replacing the injection with INJ, which incidentally proves that, actually, Csharpminor makes the invariant $\mathrm{dom}(\alpha)$ hold.

***Implementation*** To realize those axioms, we use information contained in tags to first compute the block mapping $\iota$ from the source memory $m$. For any block identifier $b$, if the tag of $b$ in $m$ is $\mathsf{Stack}(\mathsf{Main}(\ldots))$, then $\iota(b)$ is undefined; if the tag of $b$ in $m$ is $\mathsf{Stack}(\mathsf{Var}(f, id, b', sz, of))$, then $\iota(b) = (b', of)$; otherwise, $\iota(b) = (b, 0)$.

Then, we must assume that the memory $m$ is *well-formed*: for any block $b$ of $m$ of tag $\mathsf{Stack}(\mathsf{Main}(\ldots))$, this block is empty, there are no pointer to it anywhere in the memory, and it is followed by exactly the right number of blocks of tag $\mathsf{Stack}(\mathsf{Var}(f, id, b, sz, of))$ corresponding to the local variables of $f$ and whose valid offsets are located at offsets between 0 and $sz$. This well-formedness condition is actually an invariant satisfied by the source language, and it will be the domain of $\alpha$.

From such a memory, we can now construct the target memory $m'$ from the memory $m$ as follows, by scanning it from the first block. Assuming we treated all blocks between 1 and $b - 1$, we treat block $b$ and following as follows:

- if $b$ is the identifier of the next block available for allocation[6], then we are done.
- Otherwise, the block $b$ is well-defined. If $b$ is a heap block, then copy its contents (transforming pointers by $\iota$) to the target block with the same identifier $b$, and move to next block $b + 1$
- Otherwise, the block $b$ is necessarily of tag $\mathsf{Stack}(\mathsf{Main}(f, sz))$, and is empty, and its next blocks correspond to the local variables of $f$ (say that there are $n$ of those). Then, in the target memory $m'$, $b$ will have size $sz$ and receive the contents (accordingly transforming pointers by $\iota$) of the following blocks of $m$ at the offsets specified by their tags; but in $m'$, those blocks will be left empty. Then move to the next block $b + n + 1$.

Contrary to CompCert memory injections, there are no additional memory locations in the target that do not correspond to any source memory locations. This is enabled by the fact that we also add alignment constraints along with block tags to prevent alignment padding. For the sake of brevity, we do not explain this issue here.

### 7.5 Stack layout for spilling locations

Starting from Cminor, local variables are laid out together in a single stack frame. However, Cminor and further languages such as RTL still assume that there is an unbounded number of pseudo-registers for temporary variables whose addresses are not taken. To address this issue, CompCert performs register allocation in two stages by sorting out pseudo-registers into two classes of *abstract locations*, one for machine registers, another for stack slots corresponding to spilling locations. Then, those stack slots are integrated into memory by extending the stack frame through a stack layout compilation pass between two languages, Linear and Mach.

Linear and Mach are two assembly-like 3-address languages having similar instructions: memory load/store through pointers, conditionals, arithmetic operations, function calls, and access to the current stack frame. The main difference between the two languages is that instructions in Linear operate on abstract locations, whereas Mach operates on machine registers only. In particular, access to the stack frame do not perform memory load/store in Linear, but do in Mach; moreover, additional such instructions (*reloading*) have to be added into Mach code for each Linear instruction operating on a stack slot.

Conceptually, this pass adds new reachable locations into memory. To adapt this pass to $\alpha$-refinement, we thus have to take these additional spilling locations. However, it is not possible to predict their values in the semantics of Linear. In fact, in the original CompCert proof, those spilling locations are fixed by the relation between the source and target memory states. They must not change through function call, and in particular through external function call.

Our solution is to include the "spilling fragment" of the memory into external function call events. In Linear, they become of the form $\texttt{Extcall}(f, m_1, S, m_2)$ where $m_1$ (resp. $m_2$) is the Linear memory state before (resp. after) the call to $f$, and $S$ is the fragment of memory corresponding to the spilling locations and that is to be added in Mach. Then, we develop an operation $\triangleleft$ such that the Mach memory state $m_1' = m_1 \triangleleft S$ is obtained by "adding" the spilling locations into the Linear memory state. Then, for a Mach module obtained by compiling a Linear module, the Mach memory state is always of this form.

Because the spilling locations cannot be predicted in the Linear semantics, $S$ is arbitrary in Linear: the compositional semantics of a module says nothing about the spilling locations. By contrast, in Mach, $S$ can be fixed by explicitly declaring, at the level of a function call, where are the spilling locations of the current function (which are added to the spilling locations of the callers). So $S$ is chosen through the Linear-to-Mach refinement process, and those Linear traces where $S$ does not correspond to any Mach stack layout are simply dropped out.

To refactor the refinement, we split the current CompCert Linear-to-Mach proof into two simulation diagrams. We introduce a Linear' intermediate language as a reinterpretation of Linear performing stack operations into memory: there is no code transformation between Linear and Linear', but the spilling locations are already introduced in Linear'. So we can transform the Linear-to-Mach forward simulation proof into a Linear-to-Linear' lock-step backward simulation proof. Then, from Linear' to Mach, the code

[6] A memory always has finitely many blocks, and the number of blocks always increases because freed locations are never reused, so that a freed block is never actually deleted (only its locations are turned into unusable ones) and any newly allocated block is always fresh.

transformation remains the same as in CompCert (introduction of explicit spilling/reloading code), and its proof is a more straightforward forward simulation where the memory states are the same in Linear' as in Mach. The implementation and proof are currently in progress.

## 8. Related work and conclusions

Our compositional semantics is designed primarily for C-like languages, so it is not directly applicable to ML-like functional languages which have more sophisticated semantic models. C-like languages support first-class function pointers, but they do not allow function terms (e.g., $\lambda x.e$) as first-class values. C-like languages also support intensional operations such as equality test on function pointers, so it is unsound to replace one function pointer with another even if they point to functions with same observable behaviors. This allows us to use much simpler semantic objects (e.g., memory blocks with code pointers as in CompCert [14]) than sophisticated models developed for functional languages [2, 11, 1].

***Compositional trace/game semantics*** Our idea of modeling the behavior of each external function call as an $\texttt{Extcall}(f, m, m')$ event (see Sec. 4) resembles similar treatments in compositional trace or game semantics [5, 9]. Brookes's transition-trace semantics [5] models environment transitions for shared memory concurrent languages. Under Brookes's semantics, a thread's behavior is described as a set of transition traces, with each consisting of a sequence of state transition steps $(m_1, m_1') :: (m_2, m_2') :: \ldots :: (m_n, m_n')$. The gaps between consecutive steps (e.g., $m_1'$ and $m_2$, or $m_{n-1}'$ and $m_n$) signal those state transitions made by other threads in the environment. Composing two threads involves calculating all the interleavings of pairs of transition traces (one from each thread) and their stuttering and mumbling closures.

Our $\texttt{Extcall}(f, m, m')$ event also uses a pair of memory states $(m, m')$ to signal state transitions made by the environment (i.e., external calls). Our semantic linking operation (see Sec. 5) also does the "merging" of multiple event traces, but it requires more sophisticated substitutions (on behaviors) since we must also support divergence, I/O events, and reacting behaviors. It does not require stuttering and mumbling closure since we are only dealing with sequential languages. The proximity between these two approaches shows great promise toward combining these two techniques to build compositional models for concurrent C-like languages.

Ghica and Tzevelekos [9] developed a system-level semantics for composing C-like program modules. They also used external call and return events and used them to model open C-like modules and their environments. Our work can be viewed as an adaptation of their idea to the setting of compositional compiler correctness, with the goal of addressing language-independent behavior specifications that include divergence, I/O and reactive events.

***Compositional CompCert*** Concurrently with our work, Stewart *et al* [20, 3] have recently completed the development of a formally verified separate compiler for CompCert C. This is a very impressive achievement since their Coq implementation includes all 8 translation phases from CompCert Clight to CompCert x86 plus many of the optimization phases. They developed *interaction semantics* which is a protocol-oriented operational semantics of intermodule (or thread) interaction: an open module would take normal unobservable steps or make internal function calls (defined in the same module), but would "block" when calling external functions; each such "block" point is considered as an interaction point; the program will resume execution when the external function call returns. To support both vertical and horizontal composition, they have also developed a new form of "structured simulations" which extends CompCert-style memory injections with fine-grained subjective invariants and a leakage protocol.

While our `Extcall`-event-based semantics (EES) shares many similarities to Stewart *et al*'s interaction semantics (IS), they also have some significant differences. EES does not rely on any new "protocol-oriented" operational semantics, instead, it just treats external function calls as regular events, thus it can use the same trace-based behavior specifications as semantic objects. When linking two modules $\mathfrak{u}_0$ and $\mathfrak{u}_1$, our semantic linking operator $\bowtie$ (under EES) would automatically calculate the resulting semantic objects for the linked module $(\mathfrak{u}_0 \uplus \mathfrak{u}_1)$, replacing all cross-module calls between $\mathfrak{u}_0$ and $\mathfrak{u}_1$ with their corresponding behavior specifications. This leads to a very nice linking theorem (see Theorem 1 in Sec. 5): if $\mathfrak{u}_0$ and $\mathfrak{u}_1$ are two modules in the same language, linking their compositional semantics at the level of their behaviors exactly corresponds to the compositional semantics of their syntactic concatenation of the two modules. The interaction semantics (IS), on the other hand, does not attempt to "big-step" the cross-module calls between $\mathfrak{u}_0$ and $\mathfrak{u}_1$ during linking, thus it has not been able to prove the same linking theorem as we have done.

***Kripke logical relations*** Kripke Logical Relations (KLRs) [18] are designed to support horizontal composition for functional languages. They define equivalence between terms (and values) in such a way that two functions $f_1$ and $f_2$ (of same type) are equivalent if, and only if, for any two equivalent values $v_1, v_2$ of the same type, $(f_1\ v_1)$ and $(f_2\ v_2)$ are equivalent. Ahmed *et al* [1, 8] showed how to generalize KLRs to reason about higher-order states. Hur and Dreyer [10] rely on step-indexed logical relations to show how to support horizontal composition; they prove correctness of a one-pass compiler but they do not support vertical composition since step-indexed logical relations are known to be not transitive.

C-like languages support both first-class function pointers and states but they do not support first-class function terms as in most functional languages. Because C function pointers can be tested for equality, a function pointer can not be replaced by another, even if they point to functions that have same observable behaviors. This is why we can build much simpler semantic models and how our new compositional semantics can still establish the monotonicity (congruence) result of our refinement relation (Section 6, Theorem 2).

***Parametric bisimulations*** Hur et al. [11] recently proposed a promising approach that combines KLRs with bisimulations. The main idea is to abandon step-indexing but rely, instead, on coinductive simulation-based techniques (which are closer to CompCert-style simulation relations). More specifically, they propose to parameterize the local knowledge of functions with the global knowledge of external functions, and to define equivalence for open modules based on a simulation diagram over the small-step semantics of the two underlying languages of the programs. A simulation diagram can make two equivalent programs perform several steps from two equivalent states to two states corresponding to an external function call, then resume simulation upon return of such a call. This "disruption" in the flow of the simulation is analogous to our way of making the external function call explicit as a specific event in the behavior. Thus, our work can be seen as a *unary version* of their parametric bisimulations by defining a unary semantics for open modules but at the level of behaviors (independently of the small-step semantics of the underlying languages). Our way of defining the linking operator at the semantic level of behaviors avoids the need of strong typing, which makes our approach more amenable to support weakly typed C-like languages.

***Conclusions*** In this paper, we have presented a novel compositional semantics for reasoning about open modules and for supporting verified separate compilation and linking. To build compositional semantics for open concurrent programs, we plan to split our single `Extcall` event into separate call and return events. Semantics for open concurrent programs can then have interleaving

external call and return events. Semantic substitutions in our linking will be replaced by some form of "zipping" operations.

## Acknowledgments

We thank anonymous referees for their helpful comments and suggestions that improved this paper and the implemented tools. This research is based on work supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, NSF grant 1065451, and ONR Grant N00014-12-1-0478. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

## References

[1] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *Proc. 36th ACM Symposium on Principles of Programming Languages*, pages 340–353, 2009.

[2] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *Proc. 2009 ACM SIGPLAN International Conference on Functional Programming*, pages 97–108, 2009.

[3] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. In *Proc. 2014 European Symposium on Programming (ESOP'14)*, volume 8410 of *LNCS*, pages 107–127. Springer-Verlag, Apr. 2014.

[4] S. Blazy and X. Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[5] S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.

[6] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8:174–186, 1968.

[7] R. Dockins. *Operational Refinement for Compiler Correctness*. PhD thesis, Princeton University, 2012.

[8] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. In *Proc. 24th IEEE Symposium on Logic in Computer Science*, pages 71–80, 2009.

[9] D. Ghica and N. Tzevelekos. A system-level game semantics. In *Proc. 28th Conf. on the Mathematical Foundations of Programming Semantics (MFPS)*, 2012.

[10] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *Proc. 38th ACM Symposium on Principles of Programming Languages*, pages 133–146, 2011.

[11] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *Proc. 39th ACM Symposium on Principles of Programming Languages*, pages 59–72, 2012.

[12] X. Leroy. The CompCert verified compiler. `http://compcert.inria.fr/`, 2005–2013.

[13] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[14] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformation. *Journal of Automated Reasoning*, 2008.

[15] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[16] C. C. Morgan. *Programming from specifications, 2nd Edition*. Prentice Hall International series in computer science. Prentice-Hall, 1994.

[17] M. Müller-Olm. *Modular Compiler Verification – A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer, 1997.

[18] A. Pitts and I. Stark. Operational reasoning for functions with local state. In *HOOTS'98*, pages 227–274, 1998.

[19] T. Ramananandro, Z. Shao, S.-C. Weng, J. Koenig, and Y. Fu. A compositional semantics for verified separate compilation and linking. Technical Report YALEU/DCS/TR-1494, Dept. of Computer Science, Yale University, New Haven, CT, December 2014. URL: `flint.cs.yale.edu/publications/vscl.html`.

[20] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In *Proc. 42nd ACM Symposium on Principles of Programming Languages*, page (to appear), 2015.

[21] The Coq development team. The Coq proof assistant. `http://coq.inria.fr`, 1999 – 2015.

[22] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, Apr. 1971.

[23] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. 2011 ACM Conference on Programming Language Design and Implementation*, pages 283–294, 2011.