The Next 700 Compiler Correctness Theorems (Functional Pearl)

DANIEL PATTERSON, Northeastern University, USA
AMAL AHMED, Northeastern University, USA

Compiler correctness is an old problem, with results stretching back beyond the last half-century. Founding the field, John McCarthy and James Painter set out to build a "completely trustworthy compiler". And yet, until quite recently, even despite truly impressive verification efforts, the theorems being proved were only about the compilation of whole programs, a theoretically quite appealing but practically unrealistic simplification. For a compiler correctness theorem to assure complete trust, the theorem must reflect the reality of how the compiler will be used.

There has been much recent work on more realistic "compositional" compiler correctness aimed at proving correct compilation of components while supporting linking with components compiled from different languages using different compilers. However, the variety of theorems, stated in remarkably different ways, raises questions about what researchers even mean by a "compiler is correct." In this pearl, we develop a new framework with which to understand compiler correctness theorems in the presence of linking, and apply it to understanding and comparing this diversity of results. In doing so, not only are we better able to assess their relative strengths and weaknesses, but gain insight into what we as a community should expect from compiler correctness theorems of the future.

1 WHO VERIFIES THE VERIFIERS?

"Can you trust your compiler?" So began Xavier Leroy in a Journal of Automated Reasoning paper describing the CompCert C compiler[Leroy 2009], a tour de force in verified compilers that is implemented and *proved correct* using the Coq proof assistant. Compiler correctness is not a new problem: in a seminal 1967 paper, John McCarthy and James Painter, using a simple compiler for arithmetic expressions, proposed the problem of how to build a completely trustworthy compiler[1]. They wondered, as we wonder now, what happens if the programs we write do not correspond to what actually runs? In 1973, F. Lockwood Morris provided a more solid foundation for the field with a compiler correctness theorem that is essentially indistinguishable from what showed up in CompCert more than 30 years later: the theorem says the "meaning" of the program after compilation should be essentially the same as the "meaning" of the original source program—that is, the compiler should be *semantics preserving*.

But the compiler correctness theorem from Morris, proved for CompCert and in numerous other results before and after, is about compiling whole programs[2], and whole programs, unfortunately, are rare occurrences indeed in the diet of real-world compilers. Infrequent enough are C programs that do not use the C standard library, but once we broaden our tastes to higher-level languages the issues of runtimes and efficient low-level data-structure implementations completely disabuse us of the notion that correctly compiling "whole source programs" is enough.

If we are to have confidence that the code that we write is indeed the code that runs, we need theorems that address the reality of the programs we compile. Compiler correctness theorems tell

[1]In an earlier report, [McCarthy 1959] mentioned the goal of having theories for equivalences of transformations.
[2]Morris's denotational model included open terms, but the actual theorem relating the semantics was only over closed terms, as he stated they were the only programs that would actually be run.

Authors' addresses: Daniel Patterson, Northeastern University, 440 Huntington Avenue, Boston, MA, 02115, USA, dbp@dbpmail.net; Amal Ahmed, Northeastern University, 440 Huntington Avenue, Boston, MA, 02115, USA, amal@ccs.neu.edu.

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

us how the compiler will behave *provided we fulfill the assumptions of the theorem.* Traditional compiler correctness results, such as those of Morris and CompCert, assume that the inputs to the compiler are *complete* runnable programs.[3] Hence, in situations when the compiler that the theorem describes is actually used to compile *partial* programs (say, code that links against the standard library), the theorem itself no longer holds! So if we are to have a "completely trustworthy" compiler, the goal laid out for us by McCarthy and Painter, we are left wanting for better compiler correctness theorems.

And so we enter a more muddled world, without the benefit of a half century of clarity. In this world, fragments of code come from different languages and are stitched together before running, and the question of what researchers even mean by a "compiler is correct" has become the subject of some dispute, if only to the careful observer. In the last few years, there have been many fundamentally different theorems, all dubbed "compositional compiler correctness," that aim to capture correctness when compilers translate partial programs and allow linking with other code. But "who watches the watchmen?"[4] we ask, or in our context, "who verifies the verifiers?" We ask this because knowing that a theorem has been proved does not help understand how the theorem statement relates to reality. Clearly, if a theorem only applies when compiling whole programs, it cannot be relied upon when compiling code that will be linked. But if a theorem is more flexible, capturing "compositional correctness," then often figuring out what it states and whether it applies when linking with some particular code is subtle.

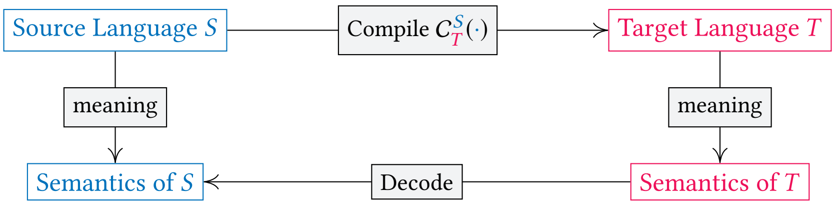
Even worse, different theorems under the umbrella of compositional compiler correctness make radically different choices that restrict, in different ways, how they can be used and what guarantees they provide. Given that compiler correctness is about trusting the behavior of the compiler, confidence that the theorem relates to its actual use is as important, if not more so, than the proof of the theorem. Because if we do not know—or cannot understand—*what* was proved, the proof itself is, perhaps, meaningless.

Our goal with this pearl is therefore threefold. First, we aim to characterize the spectrum of compiler correctness, not only for whole-program compilation, but for so-called "compositional" compilers that allow compiling partial programs and then linking them to produce what we run. Compositional compilation, we argue, is the mode of use that accurately reflects the vast majority of real-world compilation. Second, we aim to give readers a systematic methodology to assess and compare compiler correctness results as they see them. Third, we aim to hopefully convince our colleagues that having such a common compositional compiler correctness framework is useful for our research community.

2 ADVICE ON PROVING COMPILERS CORRECT

While most begin their assessment of the field of compiler correctness with McCarthy and Painter [1967], and indeed, McCarthy is usually a safe place to start, a better starting place in this case is a less cited paper by F. Lockwood Morris that appeared six years later in the first Symposium on Principles of Programming Languages (POPL) in 1973. In "Advice on Structuring Compilers and Proving Them Correct", Morris [1973] stated that his purpose was to "advise an approach" to achieve McCarthy's goal: to prove compilers completely trustworthy. He outlined his theorem with a diagram, which we approximately reproduce here:

[3]Here we mean CompCert prior to 2.7. CompCert 2.7 added support for separate compilation, incorporating the work of [Kang et al. 2016] that we describe later in this pearl.
[4]Quoting the English writer Alan Moore.



We read the diagram as follows: The source language S should be given meaning via a source semantics and the target language T should be given meaning via a target semantics, and then the compilation of S to T should relate the two semantics. Specifically, the semantics of the target program produced by the compiler should be *related* to the semantics of the source program, which is expressed via a "decode" function, since Morris concluded that the semantics of the target program may not be identical but should be somehow compatible with the semantics of the source program from which it was compiled. A typical reason we need to decode is that the target language may expose implementation details (e.g., memory allocations) that were not visible in the source semantics. This is currently more commonly referred to as *refinement*: depending on the specifics of the compiler, we say that the compiled target-language program *refines* the behavior of the source-language program, or vice versa (discussed further in §6.3). Informally, a program e_1 refines a program e_2 if every behavior of e_1 is a possible behavior of e_2.

The above theorem is thus the familiar *semantics preservation* theorem: Morris had denotational semantics in mind while CompCert (and most recent research) considers operational semantics, but the essential notion is that programs have extensional behavior—whether as mathematical functions or as running artifacts—and that this behavior should be preserved through compilation. We can define this notion more concisely: an S-to-T compiler $C^S_T(\cdot)$ is correct if

$$\forall e_S \in S.\ C^S_T(e_S)\ \ {}_T{\sqsubset}_S\ \ e_S$$

We read this as follows: if e_S is compiled from source language S to target language T, resulting in the compiled program $C^S_T(e_S)$, then the observable behavior (in the operational setting, which is our focus in this paper) of e_S is preserved in the observable behavior of $C^S_T(e_S)$—or, alternatively, the behavior of $C^S_T(e_S)$ refines the behavior of e_S. It is important to note that, throughout this pearl, whenever we write $e_1\ \ {}_{L_1}{\sqsubset}_{L_2}\ \ e_2$ we assume that e_1 and e_2 are *whole* programs—in languages L_1 and L_2, respectively—else this relation is not defined. Since the $\ \sqsubset\ $ relation is defined only on whole programs, it is clear that this theorem says absolutely nothing about what (if anything) the compiler does for partial programs.

Semantics preservation has had such staying power because the above theorem statement is so crisp: once the semantics of the source and target languages are defined, the meaning of the theorem is entirely contained in the dozen odd symbols above. While formally defining source- and target-language semantics may be non-trivial, they do not increase the burden of *understanding* whether semantics preservation implies what we want from whole-program compiler correctness. We simply expect that the source semantics must match the language definition that the programmer is using, and the target semantics must match however the programs will be run, whether that is the model of some hardware or the reduction semantics of a low-level language. There are, of course, subtleties that researchers have worked through over the years: the definition and direction of refinement as used in the statement may need to be changed in the presence of nondeterminism in one or both languages (e.g., the work by Ŝevčik et al. [2011]) or undefined behavior (see §6.3 for a more thorough discussion), but the essential character of what Morris described has served the field of compiler verification incredibly well for decades.

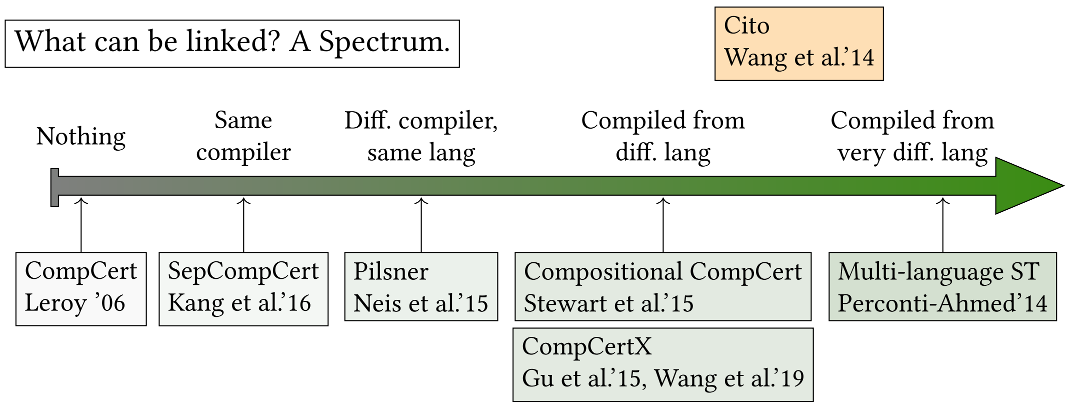


Fig. 1. A Spectrum of Linking Options

Unfortunately, the theorem is about whole programs, and it is rare to encounter real-world compilation of whole programs. Languages inevitably link against libraries after compilation, often written wholly or in part in lower-level languages. So while having whole-program compiler-correctness theorems is sure to increase the likelihood of a compiler being correct when linking,[5] it does not in any sense give us a "completely trustworthy" proof that our compilers, as used, are correct.

3 COMPILER CORRECTNESS IS A SPECTRUM

If whole-program semantics preservation, beautiful though it is, cannot account for realistic compilers, then how should our desired compiler correctness theorem be stated? We first state our goals for such a theorem:

(1) It should accommodate a suitably broad spectrum of non-whole-program compilers that exist in the world.
(2) It should accommodate existing work that has been done under the banner of compositional compiler correctness.
(3) It should be straightforward to understand.

To readers following the literature on compositional compiler correctness, achieving all three goals in a single theorem might seem to be an impossible task! We discuss the difficulties in light of the literature alongside our goals in the rest of this section.

Compiler correctness results exist on a *compositional spectrum* featuring correctness theorems that assume no linking at all, to theorems that assume linking only with code that has certain characteristics, to theorems that have incredibly broad flexibility. In Figure 1, we illustrate some points on this spectrum and where to place some recent results, characterizing the points loosely for now in terms of where the code we link with could have come from.[6] When building a verified compiler for a source language S, we can define the correctness theorem so it assumes that compiled programs will only be linked with:

[5] Yang et al. [2011]'s study found no miscompilation errors in the verified part of CompCert but many such errors in GCC and LLVM, leading them to conclude that verification of a compiler using a theorem prover has tangible benefits for users. Indeed, we have no reason to believe that that doesn't correspondingly imply that there are fewer bugs when CompCert is used to compile partial programs.
[6] In §5, when we discuss existing compiler correctness results, we will be more precise about how each result characterizes the code that the theorem supports linking with; this is usually done without reference to the origin of the code.

- code produced by the same verified compiler;
- code produced by a different compiler for the same source language S[7]
- code produced by a different compiler for a different source language R, but one with no more expressive power (in the sense of Felleisen [1990]) than S; or
- code whose behavior, ignoring performance considerations, cannot even be expressed in language S—for instance, code compiled from a more expressive source language R or written directly in a more expressive target language T.

As remarked in §1, recent compositional compiler results feature correctness theorems expressed in remarkably different ways, with the complexity of theorem statements and proofs increasing significantly once we get beyond the "from same compiler" restriction on code we link with.

Reflecting on the above, the fact that there is no single compositional compiler correctness theorem is what makes this work necessary! We want to show with this pearl not only how to arrive at a single high-level understanding of compositional correctness, analogous to Morris' for whole-program compiler correctness, but also how to reason about the diversity of meanings and attendant practical implications.

At either end of the spectrum, it is easy to understand the results and their implications: a whole-program compiler like the original CompCert [Leroy 2006] allows no linking and shows up on the far left in Figure 1, while the source-target multi-language approach of Perconti and Ahmed [2014] allows linking with arbitrary target code, even code whose functionality cannot be expressed in the source, and appears on the far right. To the right of CompCert there is SepCompCert which supports only *separate compilation* [Kang et al. 2016] and has been incorporated into the current version of CompCert. Throughout this pearl, we use the term "separate compilation" to characterize results that only support linking with code compiled by the same compiler. Next on the spectrum is Pilsner [Neis et al. 2015] where target code we link with must be related to some code in the same source language, but not necessarily produced by the same compiler. Further right, there is Compositional CompCert [Beringer et al. 2014; Stewart et al. 2015], which is not incorporated into mainline CompCert, but which is more liberal than SepCompCert in that it essentially supports linking with any code that satisfies the CompCert memory model (therefore, including code written in any of CompCert's intermediate languages). We have placed CompCertX, developed by Gu et al. [2015] and Wang et al. [2019], in the same linking category as Compositional CompCert. CompcertX supports "contextual compilation" which allows components in assembly language contexts that may not necessarily have been expressible in the source language of the compiler. [8] Finally, there is the approach taken by Wang et al. [2014], which we refer to as the "fully specified" approach, which doesn't exactly fit on the spectrum at all. This involves giving full functional specifications for the code that is being compiled and the code that will be linked, so in essence, whole-program behavior is specified before compilation. We discuss this further in §5.6.

There are many other approaches that exist at other points of the above linking spectrum or outside of it, some of which we will examine in this pearl. The essential point to note is that the question is not a binary "is this compiler compositionally correct?", but rather: what does it allow

[7] We can characterize this point in the spectrum differently: code that is provably related to (or refines) some code in language S.

[8] There are reasons why CompCertX could be placed to the left of Compositional CompCert, because, as Wang et al. [2019] write, "[Compositional CompCert] is more powerful than contextual compilation in that it allows for recursive calls between heterogeneous modules while contextual compilation does not." On the other hand, there are arguments to place it to the right of Compositional CompCert, since the assembly language contexts and lower-level memory representations that CompCertX supports are more expressive than what Compositional CompCert supports. Thus, we give up and place them at the same point in our, admittedly, imperfect spectrum.

you to link with, and what do you have to do to use it (i.e, what must you check to know that the code you wish to link with satisfies the assumptions of the theorem).

While achieving a compositional correctness theorem as simple as Morris' whole-program semantics preservation is likely impossible, we develop over the course of this pearl a common theorem outline (or framework) that is not much more complicated. We will show that for different compiler correctness results, different *parameters* we pick for the theorem will be more or less complicated, which both helps to organize the results and to compare their merits. None of this will make the theorems easier to *prove*, but we believe our contributions make it easier to understand what *has been proved*. Finally, we note that our focus is on making *theorems* easier to understand, not *proofs*. While proofs may remain the domain of experts (e.g., people working on the compiler verification project), theorems must be understandable by the masses (e.g., users of the verified compilers) if they are to provide any confidence in the trustworthiness of verified compilers.

4 ADVICE ON COMPOSITIONAL COMPILER CORRECTNESS

Consider an S-to-T compiler $C_T^S(\cdot)$ for partial programs where compiled code $e_T = C_T^S(e_S)$ can be linked with some target code e'_T—possibly produced by a different compiler, or written by hand—to produce a complete target program that can be run. We shall write ${}_L\bowtie_L$ to denote the result of linking two components (partial programs) written in language L and ${}_{L_1}\bowtie_{L_2}$ to link a component in language L_1 with a component in language L_2. Note that, in general, linking may not be defined for components from different languages; one would have to define it for specific pairs of languages. Moreover, even single-language linking is usually a partial function: it is only defined when two components can be sensibly linked together. For instance, if the language is a low-level untyped language, this might be a matter of ensuring symbols required by one component are provided by the other. If the language is typed, linking might require that the components, when put together, are well typed.

To understand how to specify compositional correctness for $C_T^S(\cdot)$, let us consider how we would use such a compiler. We would compile a partial program e_S to $e_T = C_T^S(e_S)$ and then link e_T with some e'_T to obtain a *whole* program we can run. Intuitively, for this compiler to be correct, it should be the case that the behavior of the whole program $e'_T \;{}_T\bowtie_T\; e_T$ refines the behavior of e_S linked with some code $e_?$ that is semantically equivalent to e'_T. We write $e_?$ because we aren't sure yet what language to express this program in, but we do know that no matter the language, the behavior of $e_?$ must be equivalent to e'_T. In other words, for a compiler $C_T^S(\cdot)$ to be correct, we need to know that the following holds:

$$\forall e_S \in S.\ \forall e'_T \in T.\ e'_T \;{}_T\bowtie_T\; C_T^S(e_S) \;\;{}_T\sqsubseteq_?\;\; e_? \;{}_?\bowtie_S\; e_S \qquad \text{where } e_? \approx e'_T$$

Assuming that it makes sense to link $C_T^S(e_S)$ with e'_T, and assuming that this results in a whole program, for the compiler to be correct we require a "semantics preservation" statement, as discussed for whole programs in §2. Specifically, we want the behavior resulting from linking e'_T with the source component e_S to be preserved in the result of linking e'_T with the compiled $C_T^S(e_S)$. It is critical here that linking the compiled compoenent with e'_T should yield a whole program, since our goal is to define partial-program refinement—that the partial program $C_T^S(e_S)$ refines the partial program e_S—in terms of whole-program refinement $\sqsubseteq$. Informal though it is, the above statement captures what we mean by a correct compiler: before and after compilation, it has the same behavior when interacting with e'_T. Figuring out how to make it formal, in short, how to define ? and come up with $e_?$, is the main task in front of us.

4.1 The CCC Theorem, Formally

The definition in the last section captures the intuition for when a compositional compiler should be considered correct. We now introduce the ingredients we must specify in order to state a precise compositional compiler correctness (CCC) theorem. Throughout the bulk of this pearl, we only consider linking two components, but our definitions generalize straightforwardly to an arbitrary number of components. We include the latter definition in the Appendix. We have mechanized the definitions and the proofs in §4.2 (provided as supplementary material [Patterson and Ahmed 2019]); our mechanization uses the CCC definition for an arbitrary number of components being linked together.

Linking Set. Any endeavour to formalize a compositional compiler correctness theorem must specify the set of programs that the compiler's output may be linked with. We specify this via a linking set $\mathcal{L}$, which is a set of pairs (e'_T, φ) where e'_T is a target program that the theorem deems permissible to link with and φ is a "proof" term that can be thought of as a witness to e'_T being suitable to be linked with. For example, results such as Pilsner's [Neis et al. 2015] only allow linking with target components e'_T that can be shown to be related to a source component e'_S, so any e'_T in the linking set must be accompanied by a witness φ testifying to that fact.

Source-Target Linking Medium. In the condition we sketched for compositional compiler correctness in the last section, we said that the compiled program $C_T^S(e_S)$ linked with some permissible e'_T must refine the source program e_S linked with some $e_?$ that is equivalent to e'_T, leaving open what language "?" that $e_?$ would be written in. We describe this language as the source-target linking medium $\widehat{S}$, as it is the setting in which we can meaningfully talk about linking the source component e_S with (behavior equivalent to) the target component e'_T. Some approaches to compositional compiler correctness may choose $\widehat{S}$ to be the source language S (e.g., Pilsner), in which case we need to be able to first *lift* the target term e'_T into that medium. Others may choose $\widehat{S}$ to be a mixed source-target multi-language (as in [Perconti and Ahmed 2014]), which means we can directly link e_S with e'_T.

Lift Function from $\mathcal{L}$ to $\widehat{S}$. We will require a function $\Uparrow$ (pronounced "lift") that translates T components in the linking set $\mathcal{L}$ to components in the source-target linking medium $\widehat{S}$. More precisely, the $\Uparrow$ function translates a pair (e_T, φ) from $\mathcal{L}$ to a component in $\widehat{S}$, which means that the function can make use of information in φ to generate the $\widehat{S}$ component. This will turn out to be critical for modeling results such as Pilsner and SepCompCert as we shall see in §5. In cases where the source or target include nondeterminism, defining the lift function can be subtle, which we discuss in depth in §6.2.

Linking S and $\widehat{S}$. Once we have chosen a source-target linking medium $\widehat{S}$ we will need to formally specify linking of source S components with $\widehat{S}$ components, ${}_S\!\bowtie_{\widehat{S}}$, which produces an $\widehat{S}$ term.

Linking Target Components. Finally, we need a specification of linking for target-language components: ${}_T\!\bowtie_T$. This should be a specification of the linking used by the compiler being verified.

CCC Theorem. We present the full CCC theorem below, noting that part (1) is very similar to our informal definition earlier, but made concrete using the parameters we specified above.

The subsequent requirements (2-6) are necessary to make the theorem well-behaved, and we will go through them in detail, but we present the entire definition at once to make it easier to reference. Most of the side conditions are not terribly interesting. But they are critical in ensuring that compilers that satisfy CCC are indeed compositional in the sense that we expect, as we show in the next subsection by proving that weaker compositional results follow from CCC.

Theorem 4.1 (CCC: Compositional Compiler Correctness).

$$\exists \Uparrow.\ \forall e_S \in S.\ \forall (e_T, \varphi) \in \mathcal{L}.\ e_T\ {}_T\!\bowtie_T C^S_T(e_S)\ {}_T\!\sqsubset_{\widehat{S}}\ \Uparrow\!(e_T, \varphi)\ {}_{\widehat{S}}\!\bowtie_S e_S \tag{1}$$

$$\textit{where}\quad (\varnothing_T, \varphi_\varnothing) \in \mathcal{L} \tag{2}$$

$$\forall e_S.\ \exists \varphi.\ (C^S_T(e_S), \varphi) \in \mathcal{L} \tag{3}$$

$$\Uparrow\!(\varnothing_T, \varphi_\varnothing) = \varnothing_{\widehat{S}} \tag{4}$$

$$\forall e_S.\ \varnothing_{\widehat{S}}\ {}_{\widehat{S}}\!\bowtie_S e_S\ {}_{\widehat{S}}\!\sqsubset_S\ e_S \tag{5}$$

$$\forall (e_T, \varphi) \in \mathcal{L}.\ \forall e_S.\ (\forall c_T.\ c_T\ {}_T\!\bowtie_T e_T\ {}_T\!\sqsubset_T\ c_T\ {}_T\!\bowtie_T C^S_T(e_S)) \implies (\forall c_S.\ c_S\ {}_S\!\bowtie_{\widehat{S}} \Uparrow\!(e_T, \varphi)\ {}_{\widehat{S}}\!\sqsubset_S\ c_S\ {}_S\!\bowtie_S e_S) \tag{6}$$

We walk through the theorem in detail, starting with part (1), which is the central result. Given the existence of $\Uparrow$, the theorem states that for any term e_S in S and any pair (e_T, φ) in the set of permissible-to-link terms $\mathcal{L}$, the following refinement may hold. We say "may" because the linking operation is assumed to be a partial function that may fail if a language-specific linking validation failed (e.g., a type error if the language was typed). Provided this was not the case, and the result is a whole program, the refinement is defined, and semantics preservation holds from the source term e_S linked with the lifted target $\Uparrow\!(e_T, \varphi)$ to the compiled term $C^S_T(e_S)$ linked with e_T.

It is important to describe which parts of this definition the reader must understand. Theorems are about trust, and trust is about understanding, thus the understanding burden is closely related to how trustworthy the theorem is. In this case, the reader needs to understand the lifted language $\widehat{S}$ and how linking of $\widehat{S}$ components with S terms works. On the other hand, the reader does not need to understand how the particular $\Uparrow$ function works, just that it exists (and fulfills various side-conditions to the theorem which we will go over shortly); this is why it is existentially quantified. Finally, the reader needs to understand $\mathcal{L}$, and in particular, if φ is non-trivial, will need to understand how they can be sure that the target component e_T that they want to link with is indeed something that the theorem covers.

The side-conditions of the theorem serve to ensure that the various novel parts of this definition are well-formed. Part (2) ensures that the empty component $\varnothing_T$ in the target T is in $\mathcal{L}$, which is used to prove that the definition implies whole-program correctness (a corollary we prove in §4.2).

Part (3) says that anything that is the output of the compiler is in the set $\mathcal{L}$. Note that some of these terms may not actually be able to be linked (because linking would not result in well-formed programs), but it ensures that compositionally correct compilers are correct separate compilers (another corollary we prove in §4.2).

Part (4) ensures that $\Uparrow$ translates the empty T component $\varnothing_T$ (and "empty" proof $\varphi_\varnothing$) into an empty $\widehat{S}$ component $\varnothing_{\widehat{S}}$. This seemingly trivial condition is nonetheless important in proving whole-program correctness based on CCC, and ensures that $\Uparrow$ is well-defined on empty T components.

Next, (5) ensures that linking a program e_S with an empty $\widehat{S}$ component preserves the semantics of just running e_S. This may seem trivial but note that ${}_{\widehat{S}}\!\bowtie_S$ can be instantiated to anything, and in particular there is no reason why it couldn't result in making non-trivial changes to e_S (we will see a particularly devious case of this in §6.1).

Finally, (6), requires that semantically $\Uparrow$ (lift) is the inverse of compile $C^S_T(\cdot)$ on compiler output. This may be thought of as the minimal condition required for $\Uparrow$ to be sensible (which we need since $\Uparrow$ is existentially quantified, and thus not something the user of the theorem need inspect). Hence, $\Uparrow$ and $C^S_T(\cdot)$ behave nearly as inverses: "nearly" because $C^S_T(\cdot)$ only compiles S programs but $\Uparrow$ maps to $\widehat{S}$. Specifically, this condition states that if a target component e_T (contextually) refines a compiled source component $C^S_T(e_S)$ then the lift of e_T should (contextually) refine e_S. This then has to be stated for complete programs, since $\sqsubseteq$ is whole-program refinement, which means the components need to be completed by linking with some (context) c_S or c_T.

4.2 Corollaries of The CCC Theorem

We can prove a few important corollaries of CCC. While compositional correctness is more general than whole-program correctness and separate compilation, we certainly want both of those to hold.

Corollary 4.2 (CCC Implies Whole Program Compiler Correctness).

$$CCC(C^S_T(\cdot)) \implies \forall e_S \in S.\ C^S_T(e_S)\ {}_T\!\sqsubseteq_S\ e_S$$

Proof. If we instantiate Theorem 4.1 with $\varnothing_T$ as e_T (which we know we can from Theorem 4.1 (2)), then from Theorem 4.1 (1) we get that $\varnothing_T\ {}_T\!\bowtie_T\ C^S_T(e_S)\ {}_T\!\sqsubseteq_{\widehat{S}}\ \Uparrow(\varnothing_T, \_)\ {}_{\widehat{S}}\!\bowtie_S\ e_S$.

From Theorem 4.1 (4) we know that $\Uparrow(\varnothing_T, \_)\ {}_{\widehat{S}}\!\bowtie_S\ e_S = \varnothing_{\widehat{S}}\ {}_{\widehat{S}}\!\bowtie_S\ e_S$. Further, from Theorem 4.1 (5) we know that $\varnothing_{\widehat{S}}\ {}_{\widehat{S}}\!\bowtie_S\ e_S\ {}_{\widehat{S}}\!\sqsubseteq_S\ e_S$. Since our notion of refinement is based upon set inclusion of behaviors, we can compose ${}_T\!\sqsubseteq_{\widehat{S}}$ and ${}_{\widehat{S}}\!\sqsubseteq_S$ to get that $\varnothing_T\ {}_T\!\bowtie_T\ C^S_T(e_S)\ {}_T\!\sqsubseteq_S\ e_S$. Since $\varnothing_T$ is an empty target component, linking with it does not change the behavior of $C^S_T(e_S)$, which is a whole program, and the result follows. □

Corollary 4.3 (CCC Implies Correct Separate Compilation).

$$\begin{aligned} CCC(C^S_T(\cdot)) \implies \forall e_1, e_2 \in S.\ & C^S_T(e_1\ {}_S\!\bowtie_S\ e_2)\ {}_T\!\sqsubseteq_S\ e_1\ {}_S\!\bowtie_S\ e_2 \\ \mathit{and}\quad & C^S_T(e_1)\ {}_T\!\bowtie_T\ C^S_T(e_2)\ {}_T\!\sqsubseteq_S\ e_1\ {}_S\!\bowtie_S\ e_2 \end{aligned}$$

Proof. From Corollary 4.2 we know that $C^S_T(e_1\ {}_S\!\bowtie_S\ e_2)\ {}_T\!\sqsubseteq_S\ e_1\ {}_S\!\bowtie_S\ e_2$. It remains to show that $C^S_T(e_1)\ {}_T\!\bowtie_T\ C^S_T(e_2)\ {}_T\!\sqsubseteq_S\ e_1\ {}_S\!\bowtie_S\ e_2$ as well.

We instantiate Theorem 4.1 with e_2 as e_S and $C^S_T(e_1)$ as e_T, noting from Theorem 4.1 (3) that there exists φ such that $(C^S_T(e_1), \varphi) \in \mathcal{L}$. From Theorem 4.1 (1) we get that $C^S_T(e_1)\ {}_T\!\bowtie_T\ C^S_T(e_2)\ {}_T\!\sqsubseteq_{\widehat{S}}\ \Uparrow(C^S_T(e_1), \varphi)\ {}_{\widehat{S}}\!\bowtie_S\ e_2$.

From Theorem 4.1 (6) (taking e_T to be $C^S_T(e_1)$, e_S to be e_1, and c_S to be e_2), we know that $\Uparrow(C^S_T(e_1), \varphi)\ {}_{\widehat{S}}\!\bowtie_S\ e_2\ {}_{\widehat{S}}\!\sqsubseteq_S\ e_1\ {}_S\!\bowtie_S\ e_2$, and we can compose those two together to get what we need, that $C^S_T(e_1)\ {}_T\!\bowtie_T\ C^S_T(e_2)\ {}_T\!\sqsubseteq_S\ e_1\ {}_S\!\bowtie_S\ e_2$. □

5 RESULTS FROM ACROSS THE SPECTRUM

The essential intuition of Morris' compiler correctness theorem is that for a compiler to be correct, it must be semantics preserving—that is, the behavior of the compiled code $e_T = C^S_T(e_S)$ must be related[9] to the behavior of the source e_S. While Morris' theorem formalizes semantics preservation for whole-program compilation, we note that CCC, in essence, does the same for partial-program compilation, with the main difference being that defining relatedness for whole programs is easier than defining relatedness for partial programs (which we denote $e_T\ {}_T\!\approx_S\ e_S$).

Assuming one has a means of specifying ${}_T\!\approx_S$, we should always be able to state compositional compiler correctness as $C^S_T(e_S)\ {}_T\!\approx_S\ e_S$. The central difference between existing compositional compiler correctness results arises due to different approaches to defining the relation $C^S_T(e_s)\ {}_T\!\approx_S\ e_S$. All existing results define this partial-program relation by considering all target components e'_T that when linked with $C^S_T(e_s)$ produce a whole program $e'_T\ {}_T\!\bowtie_T\ C^S_T(e_s)$. This approach is taken because it means we can rely upon the much easier-to-define relation for whole programs; in the operational setting, we can simply require that the compiler preserves the behavior observed by *running the program*, something that is clearly impossible to do of a partial program. Then each

[9]We use the term relation as a high-level notion to capture both equivalence relations and the refinements that occur in some work.

compiler correctness result ends up having to define some version of the program $e_? \; {}_?\!\bowtie_S \; e_S$ for the other side, where $e'_T \; {}_T\!\approx_? \; e_?$. That is, we must come up with some source-like equivalent of that target module such that we can construct a whole program by linking it with e_S. The theorems for these results then ensure that the behavior of the complete program $e'_T \; {}_T\!\bowtie_T \; C^S_T(e_s)$ is related to the behavior of $e_? \; {}_?\!\bowtie_S \; e_S$ (which the user of the theorem must understand, along with the relation $e'_T \; {}_T\!\approx_? \; e_?$). These insights are precisely the ones used when motivating our CCC theorem in §4.

In this section, we work our way across the spectrum of compositional compiler correctness, examining how existing results define the relation $e_T \; {}_T\!\approx_S \; e_S$, and showing how each result's position on the linking spectrum can be understood using our CCC theorem. In doing so, we will reveal various features of the approaches taken in the literature. While these are not the only pieces of prior work on compositional compiler correctness—in particular, we do not examine the most recent results by Wang et al. [2019] and Jiang et al. [2019]—they represent a suitably broad cross-section. At the end of this section, we will discuss an approach that does not fit as neatly into the CCC framework.

5.1 Linking with code generated by same compiler: SepCompCert

CompCert, an optimizing C compiler developed within the Coq proof assistant by Leroy et al. for over a decade, has been by any measure an enormous research success. Not only has it seen adoption within high-assurance areas of industry, but there have been countless extensions of the work, from within and outside the core group at Inria. However, a longstanding issue was that the correctness theorem for CompCert only held for whole programs. At the same time, perhaps dangerously, the CompCert compiler, extracted from Coq, did not have that restriction:

> "While the soundness proof for Compcert [sic] does not account for separate compilation and assumes that whole programs are compiled at once, the compiler can be used to separately compile C source files and link them with precompiled libraries, which is convenient for testing." — Leroy [2009]

During the decade after the first paper was released, various people worked on the problem of supporting linking, using different approaches, but the result that ended up incorporated into CompCert 2.7 was the work by Kang et al. [2016], which they titled SepCompCert. SepCompCert is interesting in that it explicitly "aimed lower" than existing work at the time, extending CompCert by only supporting linking with output that was produced by the exact same compiler, in an effort to reduce the amount of changes to the verified compiler. Interestingly, the effort revealed bugs in CompCert—in particular, one that only showed up in the presence of separate compilation. The latter re-iterates the necessity of having correctness theorems that apply to the reality of how compilers are used. Because while CompCert since the beginning could be used as a separate compiler—and even now, with the incorporation of SepCompCert, can be used to link against arbitrary assembly—doing so gives up the guarantees that a verified compiler comes with because such linking violates the assumptions of the correctness theorem.

Now, given that SepCompCert only allows linking with output that was generated by the same compiler, and thus is really just a separate compilation result, rather than "truly compositional", the careful reader might cry foul that this means that CCC does not, indeed, imply compositional compiler correctness, if a compiler that is not truly compositional can be shown to satisfy it. Indeed, the critique is somewhat valid, but the key to the CCC approach is the way in which the parameters to the theorem—what we are allowed to link with, the source-target linking medium, etc.—illuminate the result itself. Looking at the linking set $\mathcal{L}$ for SepCompCert (see below), it's immediately clear that this is a separate compilation result. In this way, while the very fact that a compiler can fit into the CCC framework does not imply any particular flavor of compositional correctness, the point that we will show the reader by the end of this section is that no two results agree on a particular flavor either. Therefore, the true value of a theorem like CCC is that it illuminates the questions that one should have, including what can and cannot be linked. So perhaps the reader is right and we should have called our theorem just CC, and done away with "compositional" from the name. But we plead our defense that without the important questions brought up by compositionality, Morris's theorem from 1973 would have been just fine, and so highlighting the importance of the spectrum of compositional compiler correctness in the theorem title itself is worth a little lack of precision.

For SepCompCert, as for each of the results we consider, we first present a summary of how the CCC theorem is instantiated, then describe in detail each point, and finally explain why the theorem Kang et al. prove implies our CCC theorem.

$\mathcal{L}$	$\{(e_T, \varphi) \mid \varphi =$ source component e_S that was compiled by the SepCompCert compiler to $e_T\}$
$\widehat{S}$	unchanged source language S
${}_{\widehat{S}}\bowtie_S$	unchanged source language linking ${}_S\bowtie_S$
${}_{\widehat{S}}\sqsubseteq_S$	source language (whole program) refinement ${}_S\sqsubseteq_S$
$\Uparrow(\cdot)$	$\Uparrow(e_T, e_S) = e_S$

We can see that in this case, the language $\widehat{S}$ is actually just the source language S. This is the reason for much of the simplification that characterized the SepCompCert effort (and why it was incorporated into CompCert, unlike the more powerful and earlier Compositional CompCert result by Stewart et al. [2015]), as it means that much of the rest of the complexity (the cross-language linking, the operational semantics of running $\widehat{S}$ programs) evaporates. What is left is our lift function, but this too becomes simple, because of the SepCompCert restriction that we only link with components (i.e., modules) that were produced by the SepCompCert compiler. We can realize that restriction by having the evidence φ that a component e_T is permissible to link with be just the source component e_S that was compiled to produce e_T. This then means that the lifting function is trivial, as it can simply return this component. In this, we can see the various parts of the CCC theorem fit together.

Next, we have to show that the theorem is implied by the SepCompCert correctness theorem. Given the instantiations above (which we use to specialize the theorem), we must now show:

$$\forall e_S \in S.\ \forall (C_T^S(e'_S), e'_S) \in \mathcal{L}.\ C_T^S(e'_S)\ {}_T\bowtie_T\ C_T^S(e_S)\ \ {}_T\sqsubseteq_S\ \ e'_S\ {}_S\bowtie_S\ e_S \tag{1}$$

$$\begin{aligned}
\text{where}\quad & (\varnothing_T, \varphi_\varnothing) \in \mathcal{L} && (2)\\
& \forall e_S.\ (C_T^S(e_S), e_S) \in \mathcal{L} && (3)\\
& \Uparrow(\varnothing_T, \varphi_\varnothing) = \varnothing_S && (4)\\
& \forall e_S.\ \varnothing_S\ {}_S\bowtie_S\ e_S\ \ {}_S\sqsubseteq_S\ \ e_S && (5)
\end{aligned}$$

$$\begin{aligned}
\forall (C_T^S(e'_S), e'_S) \in \mathcal{L}.\ \forall e_S.\ (\forall c_T.\ c_T\ {}_T\bowtie_T\ C_T^S(e'_S)\ \ {}_T\sqsubseteq_T\ \ c_T\ {}_T\bowtie_T\ C_T^S(e_S)) \implies \\
(\forall c_S.\ c_S\ {}_S\bowtie_S\ \Uparrow(C_T^S(e'_S), e'_S)\ \ {}_S\sqsubseteq_S\ \ c_S\ {}_S\bowtie_S\ e_S)
\end{aligned} \tag{6}$$

The side-conditions mostly follow easily, as $\varnothing_S$ compiles to $\varnothing_T$ (i.e., an empty component compiles to an empty component) covering (2) and (4), SepCompCert permits linking with components compiled by it, covering (3), and (5) follows since $\sqsubseteq$ is reflexive. The only condition that's a little tricky is the last one, where essentially, we need to show that compilation (of components) preserves refinement. But this is exactly implied by the correctness properties that SepCompCert proves (both the level A correctness in §2.2 and level B correctness in §2.3 of [Kang et al. 2016]).[10]

Finally, we must show that (1) holds. But given the concrete instantiation the result is again the statement of separate compilation.

Thus, we can see that SepCompCert fulfills a CCC theorem instantiated as above, and so fits into our framework.

5.2 Linking with code representable in the source language: Pilsner

For their compositional correctness result for Pilsner, Neis et al. [2015] define $e_T \ {}_T{\approx}_S \ e_S$ using a *cross-language relation*. The relation specifies when components from language T should be considered behaviorally related to components from language S in a manner similar to that of *cross-language logical relations*. The latter have been used for years to establish the correctness of transformations—e.g, for correctness of typed closure conversion [Minamide et al. 1996], correctness of a multi-pass compiler for STLC [Chlipala 2007]. More recently, Benton and Hur [2009] used a step-indexed logical relation [Ahmed 2006; Ahmed et al. 2009] to prove correct compilation from STLC with recursion to an SECD machine, and later Hur and Dreyer [2011] did the same for compilation from an idealized ML to assembly, both single-pass compilers because it was unclear how to scale the cross-language logical-relations approach to multi-pass compilers. Assume a compiler from source language S to intermediate language I and from I to target language T. The key technical difficulty was proving transitivity in the presence of state —that $e_T \ {}_T{\approx}_I \ e_I$ and $e_I \ {}_I{\approx}_S \ e_S$ implies $e_T \ {}_T{\approx}_S \ e_S$—which is a property we need in order to compose correctness lemmas for successive compiler passes into a single theorem that end-to-end multi-pass compilation is correct. Neis et al. [2015] addressed this by developing PILS (parametric inter-language simulations) that can be proved transitive and used PILS between S and I, and I and T to prove a compositional correctness theorem for Pilsner.

An issue with all cross-language-relation approaches to compiler correctness is that if we compile e_S—so we have $C^S_T(e_S) \ {}_T{\approx}_S \ e_S$—and link the compiled code with some e'_T, we are required to show that there exists some $e'_S \in S$ such that $e'_T \ {}_T{\approx}_S \ e'_S$. If the latter is true then we can conclude that the behavior of $C^S_T(e_S)$ linked with e'_T is related to the behavior of e_S linked with e'_S. But finding such a source component is challenging, unless the e'_T that we wish to link with is the result of compiling e'_S using the *same* compiler—which yields a proof of $e'_T \ {}_T{\approx}_S \ e'_S$—or using a different S-to-T compiler verified using the *same* cross-language relation ${}_T{\approx}_S$. Even if it were possible to easily find these examples, this still restricts linking to target components e'_T that are representable in the source language S.

Hence, we can see that cross-language approaches provide a generalization of a separate compilation result. In theory, one can link with any target code e'_T that can be shown to be related to some source component, but in practice this will only be feasible if that source component had been compiled by the same compiler—or a different compiler from the same source to the same target that was verified using the *same* source-target cross-language relation! Unless one of the latter is true, meaning there is a compiler that can produce a proof that e'_T is related to some source component, the cross-language approach comes with a pretty significant amount of formalism that compiler users must understand since without understanding the source-target relation, they can't understand how to ensure that any linking they do with some e'_T is deemed permissible by the theorem!

[10]Note that SepCompCert's level A correctness assumes that the components being linked together have been compiled by exactly the same passes, whereas level B correctness is more relaxed: it allows for some intra-language RTL optimizations to be skipped when compiling some of the linked modules.

The particular compiler that Neis et al. [2015] proved correct was a two-pass compiler from an ML-like language through an untyped intermediate language to an idealized assembly. As before, we first present the instantiation and then go through each part in detail. Finally, we show how the work done by the authors proves the instantiated CCC theorem. In this case, we are considering either a single pass or the entire compiler; since they prove transitivity on both passes, from our perspective these are similar (we discuss transitivity in much more depth later).

$\mathcal{L}$	$\{(e_T, \varphi) \mid \varphi = \text{source component } e_S \text{ and a proof that } e_S \text{ is PILS-related to } e_T\}$
$\widehat{S}$	unchanged source language S
${}_{\widehat{S}}\bowtie_S$	unchanged source language linking ${}_S\bowtie_S$
${}_{\widehat{S}}\sqsubset_S$	source language (whole program) refinement ${}_S\sqsubset_S$
$\Uparrow(\cdot)$	$\Uparrow(e_T, (e_S, \_)) = e_S$

First, we can see that as with SepCompCert, the language $\widehat{S}$ is the unadulterated source language S. What differs is the definition of the linking set $\mathcal{L}$. In this sense, Pilsner generalizes separate compilation by allowing not just a target term e_T that was compiled from a source term e_S by the Pilsner compiler, but rather any target that is PILS-related to some source term. In the Pilsner paper (a detailed description is beyond the scope of this pearl), they set up a cross-language Parametric Inter-Language Simulation (PILS) relation that allows them to formally prove source and target terms related. They then prove, in particular, that their compiler produces PILS-related target terms, so that this compiler will indeed be a correct separate compiler. But, the compiler is not limited to separate compilation: they demonstrate that Pilsner code can be linked with the output of a second compiler, Zwickel, that is verified using the same source-target PILS relation. They also show an example where they have a hand-written assembly component shown PILS-related to a source term (and thus a potential target for linking). However, all target code that can be linked must be related to some source code. And indeed, part of the process of showing that a target component is related to some source component can be seen as compiling, by hand, some source component. Essentially, Pilsner generalizes across many different PILS verified compilers, allowing separate compilation of modules using a heterogeneous set of compilers. There is a question as to the practicality of such a scheme, as the reader may wonder how likely it would be to have multiple PILS-certified compilers from the same source to the same target, with each one using precisely the same source-target PILS relation. Assuming that this is unlikely, the simpler approach of SepCompCert may be preferable.

As before, we must show that, instantiated with the above, the CCC theorem holds based on what Neis et al. [2015] have shown about their compilers.

$$\forall e_S \in S.\ \forall (e_T, (e'_S, \_)) \in \mathcal{L}.\ e_T\ {}_T\bowtie_T\ C_T^S(e_S)\ \ {}_T\sqsubset_S\ \ e'_S\ {}_S\bowtie_S\ e_S \tag{1}$$

$$\begin{aligned} \text{where}\quad & (\varnothing_T, \varphi_\varnothing) \in \mathcal{L} & (2)\\ & \forall e_S.\ \exists p.\ (C_T^S(e_S), (e_S, p)) \in \mathcal{L} & (3)\\ & \Uparrow(\varnothing_T, \varphi_\varnothing) = \varnothing_S & (4)\\ & \forall e_S.\ \varnothing_S\ {}_S\bowtie_S\ e_S\ \ {}_S\sqsubset_S\ \ e_S & (5)\end{aligned}$$

$$\begin{gathered}\forall (e_T, (e'_S, p)) \in \mathcal{L}.\ \forall e_S.\ (\forall c_T.\ c_T\ {}_T\bowtie_T\ e_T\ \ {}_T\sqsubset_T\ \ c_T\ {}_T\bowtie_T\ C_T^S(e_S)) \implies \\ (\forall c_S.\ c_S\ {}_S\bowtie_S\ e'_S\ \ {}_S\sqsubset_S\ \ c_S\ {}_S\bowtie_S\ e_S)\end{gathered} \tag{6}$$

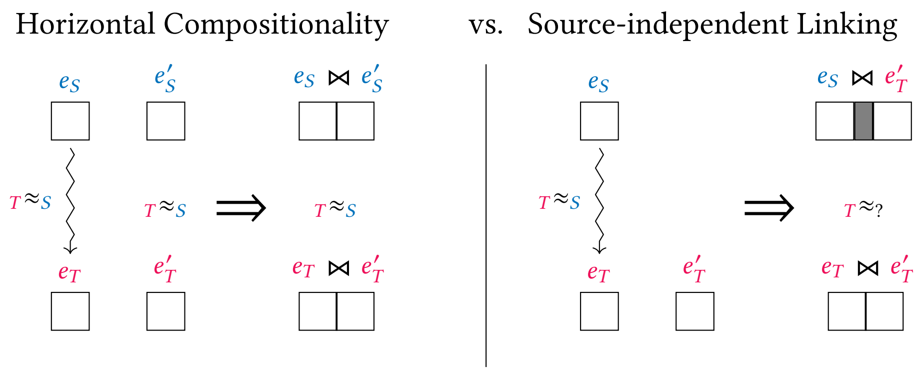


Fig. 2. Compositional verified compilers should support Source-Independent Linking

In this case, the result follows for essentially the same reason as for SepCompCert (relying in this case on Theorems 1 & 2 from §2.1 of [Neis et al. 2015]), as the only difference is in the form of $\mathcal{L}$, and with that restriction the (essentially) separate compilation result from Pilsner is sufficient.

Digression: Horizontal Compositionality vs. Source-independent Linking

In the literature on compositional compiler correctness, a compiler is said to support *horizontal compositionality* if it has support for *linking*—that is, the two terms are conflated. We think this conflation is detrimental and that it's important to draw a distinction between the two.

What the phrase "horizontal compositionality" should mean—appealing to how it's traditionally used in the PL community—is what the left side of Figure 2 depicts: that if we have a source component e_S that is related to a target component e_T (i.e., $e_T \ {}_T{\approx}_S \ e_S$), and similarly for e'_S and e'_T, then the composition (linking) of e_S and e'_S should be related to the composition of e_T and e'_T.[11] Note that a compiler that supports horizontal compositionality as depicted in Figure 2 compiles a component e_S to a component e_T and then allows e_T to be linked with a target component e'_T *provided* there exists some e'_S that corresponds to e'_T. (This is precisely the kind of linking that Pilsner and SepCompCert support.)

We argue that horizontal compositionality constrains us to a highly impractical linking scenario, and that compositionally verified compilers should aspire to do better. Contrast horizontal compositionality with what we call "source-independent linking", depicted on the right in Figure 2. Here, we allow linking with some target code e'_T without having to exhibit some source term e'_S that it is related to. Since there is no natural operational notion of directly linking source and target code, the formalism must provide some "adaptor blob", which we portray here shaded. Note, in particular, this means it may be possible to link with target code for which there is no equivalent source code, as often happens in realistic settings where libraries are written in lower-level languages in order to get access to features that don't exist in the high-level language—such as threading, randomness, etc. To be correct in this setting, the compiler verification effort must provide some interpretation of linking the source term e_S with the target of linking e'_T, and then show that this is equivalent to

[11]This is the kind of compositionality supported by logical relations, which we've discussed have long been considered useful for proving correctness of translations. We believe the idea that horizontal compositionality gives us support for linking became popular following Benton and Hur [2009], though they referred to the property simply as compositionality. The adjective "horizontal" was added later to distinguish it from a "vertical" compositionality property needed when verifying multi-pass compilers [Perconti and Ahmed 2014; Ramananandro et al. 2015; Stewart et al. 2015].

e_T linked with e'_T. In the diagram, we wrote ? for the language that results from linking e_S with e'_T, but within CCC, this is exactly the role of the $\widehat{S}$ language.

Horizontal compositionality is not the right property for the linking scenarios supported by real-world compilers. Instead our compiler correctness results should strive to support source-independent linking. Indeed, the CCC theorem provides a way of allowing this flexibility while still making the resulting theorem accessible, and thus useful.

The next two compiler correctness results we discuss, the multi-language approach from Perconti and Ahmed [2014] and Compositional CompCert from Stewart et al. [2015], both support source-independent linking.

5.3 Linking with code from an arbitrary language: Source-Target Multi-language

In 2007, Matthews and Findler [2007] came up with the idea of a syntactic multi-language as a way of describing two languages interacting; in their case, a typed ML-like language and an untyped Scheme-like language. When designing a multi-language, we merge the syntax of both languages and then add special boundary terms to mediate between the two languages. For languages S and T, the boundary $\mathcal{ST}(\cdot)$ allows T terms to be embedded in S contexts (of appropriate type, if the languages are typed), and the boundary $\mathcal{TS}(\cdot)$ allows S terms to be embedded in T contexts (of appropriate type). In the formulation by Matthews and Findler [2007], this was a notion entirely about different source languages interacting, but Ahmed and Blume [2011] realized that if applied to the source and target of a compiler, it gave a clear answer to the question posed by CCC: how to reason about the interactions of the target library to be linked after compilation and the source component before compilation. Once you have such a source-target multi-language, linking target components and source components is a native feature. Ahmed and Blume built and verified a simple compiler that did a continuation-passing-style transformation, proving as the central result that the compiled code is equivalent, in the multi-language, to the source component wrapped in a boundary: $C_T^S(e_S) \approx \mathcal{TS}(e_S)$, which means that using these components in any way within the multi-language always produces the same result.

In the context of CCC, we consider the later work by Perconti and Ahmed [2014], which used the same multi-language technique to verify a type-preserving, multi-pass compiler from System F through closure conversion (language C) and heap allocation (language A). Using the multi-language, this compiler allows linking with arbitrary target code of translation type τ^+ (where $(\cdot)^+$ is a type-translation function), including code that is not expressible in the source, because correctness is established component-wise in the multi-language. Indeed, in the example used the source language is System F with existentials and recursive types but the target is a stateful language, and we can link with arbitrary target code (provided it typechecks). In order to support the two passes, they define a multi-language FCA that embeds all three languages used within the compiler. Compiler correctness is then phrased as observational equivalence within the combined FCA multi-language. In particular, the authors highlight that this allows linking with arbitrary target code, even code that is not representable in the source.

As before, we show the instantiation of the CCC framework, describe each aspect, and then show that the CCC theorem holds based on the work in Perconti and Ahmed [2014]. Below we simply refer to the languages as S and T (with boundaries $\mathcal{ST}(\cdot)$ and $\mathcal{TS}(\cdot)$), but S and T are actually F and C for the first pass of Perconti and Ahmed [2014]'s compiler (with boundaries $\mathcal{FC}(\cdot)$ and $\mathcal{CF}(\cdot)$), and C and A for the second pass (with boundaries $\mathcal{CA}(\cdot)$ and $\mathcal{AC}(\cdot)$).

$\mathcal{L}$	$\{(e_T, \_) \mid$ where e_T is any target component of some translation type $\tau^+\}$
$\widehat{S}$	source-target multi-language ST with boundaries and source-target value translation
$_{\widehat{S}}\bowtie_S$	$e' \;{}_{ST}\bowtie_{ST}\; e_S$
$_{\widehat{S}}\sqsubset_S$	runs $\widehat{S}$ according to multi-language ST operational semantics, compares with running S
$\Uparrow(\cdot)$	$\Uparrow(e_T, \_) = \mathcal{ST}(e_T)$

One thing that the reader should immediately note is that, unlike the previous examples, the multi-language approach makes no (a priori) restriction on what we can link with—there will, of course, be components that produce errors when linking, but that's based on the definition of the language and its type system, rather than a restriction of the compiler verification. This is reflected in the composition of the linking set $\mathcal{L}$. On the other hand, the lifted language $\widehat{S}$ is an ST multi-language, which includes all of the behavior of both the source language S and the target language T. Note that we have (arbitrarily) chosen that cross-language linking within the multi-language will take place with S components at the boundary. This means that cross-language linking will pass e_S through unchanged, whereas the lifting function $\Uparrow$ will wrap the target term in a boundary.

Given that instantiation, we present the theorem that we must show to follow from the work of Perconti and Ahmed [2014]:

$$\forall e_S \in S.\ \forall (e_T, \_) \in \mathcal{L}.\ e_T \;{}_T\bowtie_T\; C_T^S(e_S) \;\;{}_T\sqsubset_{ST}\;\; \mathcal{ST}(e_T) \;{}_{ST}\bowtie_{ST}\; e_S \tag{1}$$

$$\text{where}\quad (\varnothing_T, \varphi_\varnothing) \in \mathcal{L} \tag{2}$$

$$\forall e_S.\ (C_T^S(e_S), \_) \in \mathcal{L} \tag{3}$$

$$\Uparrow(\varnothing_T, \_) = \varnothing_{ST} \tag{4}$$

$$\forall e_S.\ \varnothing_{ST} \;{}_{ST}\bowtie_S\; e_S \;\;{}_{ST}\sqsubset_S\;\; e_S \tag{5}$$

$$\begin{aligned}\forall (e_T, \_) \in \mathcal{L}.\ \forall e_S.\ &(\forall c_T.\ c_T \;{}_T\bowtie_T\; e_T \;\;{}_T\sqsubset_T\;\; c_T \;{}_T\bowtie_T\; C_T^S(e_S)) \implies \\ &(\forall c_S.\ c_S \;{}_{ST}\bowtie_{ST}\; \mathcal{ST}(e_T) \;\;{}_{ST}\sqsubset_S\;\; c_S \;{}_S\bowtie_S\; e_S)\end{aligned} \tag{6}$$

As before, the only illuminating side-conditions are (3), (5), and (6). For (3), note that this holds trivially, since all target terms of translation type are in $\mathcal{L}$. For (5), note that what this requires we show is that $\mathcal{ST}(\varnothing_T) \;{}_{ST}\bowtie_S\; e_S \;\;{}_{ST}\sqsubset_S\;\; e_S$, but this follows from the fact that the multi-language runs a purely S term according to the S operational semantics.

For (6), we must show that for any component e_S such that the semantics of $C_T^S(e_S)$ are preserved in e_T, the semantics of e_S are preserved in $\mathcal{ST}(e_T)$. Since the multi-language is a congruence, it follows from the premise that the semantics of $\mathcal{ST}(C_T^S(e_S))$ are preserved in $\mathcal{ST}(e_T)$. Then, to complete the proof, we rely on the proof of compiler correctness (§7.2 in [Perconti and Ahmed 2014]), which tells us that $e_S \approx \mathcal{ST}(C_T^S(e_S))$.

Finally, we must show that the central result (1) holds. Here, we run the T program on the left and compare with running the program on the right wrapped in $\mathcal{TS}(\cdot)$. By multi-language boundary cancellation, the program on the right is equivalent to $e_T \;{}_{ST}\bowtie_{ST}\; \mathcal{TS}(e_S)$. By compiler correctness (§7.2 in Perconti and Ahmed [2014]) we have $e_S \approx \mathcal{ST}(C_T^S(e_S))$, or equivalently by boundary cancellation, that $\mathcal{TS}(e_S) \approx C_T^S(e_S)$. From this equivalence of components, it follows that they can be linked with e_T to yield equivalent programs.

5.4 Linking with languages with common values and interaction protocol: Compositional CompCert

Since the first publication in 2006, CompCert has been extended by many groups in many ways, but for nearly a decade papers would either explicitly acknowledge leaving non-whole-program compilation to future work, or perhaps simply ignore the issue. Before SepCompCert, which we discussed in §5.1, came another result, Compositional CompCert [Stewart et al. 2015], described by the authors as "the first verified separate compiler for C". But while Stewart et al. [2015] did build a separate compiler in Compositional CompCert, they actually did quite a lot more. The novel technique they followed involved defining an interaction protocol to support "language independent-linking", an interaction semantics that would support any language that obeyed the protocol. While it was independent of the particular language, the protocol as specified did require that the languages satisfy the CompCert memory model. The interaction semantics thus allowed components to be written in any of the CompCert languages (all of which support the same memory model), and probably any similar languages, which is a much more flexible notion of correctness than that supported by SepCompCert, the subsequent work that eventually made it into the official CompCert compiler.

What is interesting about Compositional CompCert is the similarity with Perconti and Ahmed [2014]'s (syntactic) multi-language approach: Compositional CompCert essentially defined a *semantic* multi-language with its notion of language-independent linking. The relation ${}_{T}{\approx}_{S}$ that Compositional CompCert uses to establish the correctness of each pass is a contextual equivalence over modules that they defined in the context of their interaction semantics. Like Perconti and Ahmed [2014], Compositional CompCert also marked a break from the line of work that Pilsner had extended, because it made no requirements that linked code be expressible in the source language. In the end, Compositional CompCert was a significant change for CompCert, which is what motivated the much simpler SepCompCert development that (nonetheless) produced the proof of separate compilation. But separate compilation is not a universal solution, and understanding what Compositional CompCert did and how it compares is useful for future work on verified compilation.

As before, we interrogate the formalism by way of instantiation using the CCC theorem:

$\mathcal{L}$	$\{(e_T, \_) \mid$ where e_T is any target component $\}$
$\widehat{S}$	a syntax that embeds source and target, equipped with interaction semantics
${}_{\widehat{S}}{\bowtie}_{S}$	adding another module to combined syntax
${}_{\widehat{S}}{\sqsubset}_{S}$	runs $\widehat{S}$ according to interaction semantics, compares trace with running S
$\Uparrow(\cdot)$	$\Uparrow(e_T, \_) = e_T$

Note that in this setting, we have had to do more than in previous examples, because the Compositional CompCert formalization does not have a syntactic representation of multi-language programs, instead defining them directly in terms of the Coq code in which they are formalized. In this way, our $\widehat{S}$ does not exist in the syntactic sense in the work presented by Stewart et al. [2015], unless we take $\widehat{S}$ to be Coq itself.

In this case, the CCC theorem holds for much the same reason as it holds for the multi-language approach (particularly relevant for the proof is Theorem 2 from §5 of [Stewart et al. 2015]).

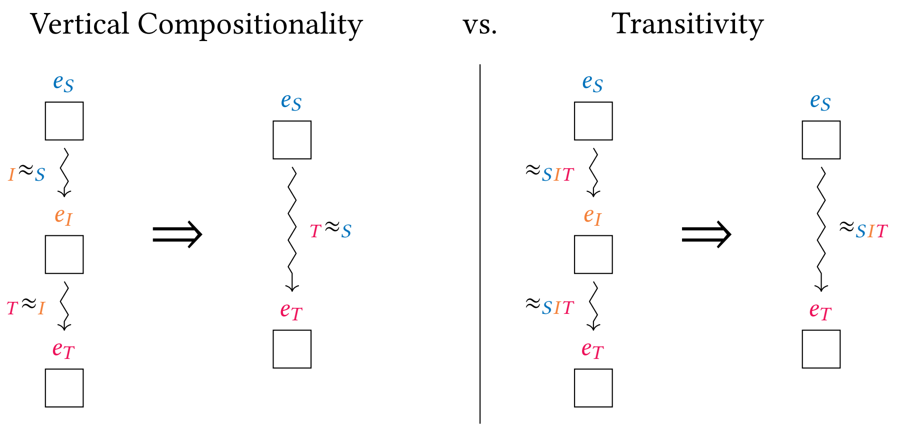


Fig. 3. Vertical Compositionality is Better than Transitivity

Digression: Vertical Compositionality vs. Transitivity

In the compositional compiler-correctness literature, a compiler with a pass from language S to I and another pass from I to T is said to support *vertical compositionality* whenever it supports a *transitivity* property: if e_T and e_I are related and e_I and e_S are related, then e_T and e_S are related. As discussed in the literature, vertical compositionality or transitivity is critical for modular (one pass at a time) verification of multi-pass compilers, so we can prove each pass correct and compose the correctness theorems into an end-to-end compiler correctness theorem for the entire compiler.

We think it is detrimental to conflate vertical compositionality with transitivity. What the phrase "vertical compositionality" should mean is what the left side of Figure 3 depicts: that we should be able to verify each pass of the compiler in isolation (using the relation $_I{\approx}_S$ relation for the first pass and $_T{\approx}_I$ for the second pass), without taking into consideration what optimizations other passes might perform or what intermediate languages we might find a few more levels higher or lower in the compiler pipeline.

Contrast vertical compositionality on the left of the figure with how Perconti and Ahmed [2014] achieve transitivity for their two-pass verified compiler, depicted on the right in Figure 3. They merge all the languages in the compiler pipeline into a single multi-language, denoted SIT in the figure, and then use the same SIT contextual equivalence to establish the correctness of each pass of the compiler. Contextual equivalence is transitive, so they get transitivitiy for free. But since all the languages in the compiler pipeline are taken into account when verifying each pass, this is far less modular than vertical compositionality.

Compositional CompCert [Stewart et al. 2015] also fails the true vertical compositionality test, though in their case it's less obvious. Specifically, they use structured simulations to verify the correctness of each pass. In order to define these structured simulations, they must take into account all of the memory transformations that can possibly happen in *any* pass of the compiler. Some compiler passes do memory extensions while others do memory injections. Their structured simulations must be specified to impose a single rely-guarantee protocol between all the possible languages in the compiler pipeline. This task is made easier for them, as compared to the multi-language approach, because all of their compiler languages make use of the same CompCert memory model, whereas the multi-language approach is able to deal with languages with different memory models. Still, conceptually, both Compositional CompCert and the multi-language approach achieve transitivity in a technical sense, by "pulling" all of their compiler languages into a single "model" (interaction semantics or multi-language semantics) and then using this single model to prove the correctness of every single pass of the compiler. Thus, they achieve transitivity in a technical sense and can therefore build multi-pass compilers, but neither support true vertical compositionality.

Finally, both approaches suffer from a similar weakness: if we extend either compiler with an additional pass, we may have to reprove the correctness of all existing passes in the compiler. Specifically, for the multi-language approach, if we add a pass to a new target language U that is more expressive than the multi-language SIT, then extending the multi-language with the new language U might break equivalences that held in SIT. That could, in turn, invalidate correctness of existing optimizations and transformations. Similarly, in the case of Compositional CompCert, if we add a pass that performs a new memory transformation that requires a change to the structured simulation, we may have to redo proofs of existing passes.

In comparison, Pilsner actually does support true vertical compositionality: they show that if $e_T \;{}_T{\approx}_I\; e_I$ and $e_I \;{}_I{\approx}_S\; e_S$, then $e_T \;{}_T{\approx}_S\; e_S$, though this requires significant effort to prove.

We argue that compiler correctness results should aim for vertical compositionality as depicted on the left side of the figure—that is, they should try to make use of modular proof architectures that allow every pass of the compiler to be verified without having to think about what languages or transformations are happening in other parts of the compiler. Such compilers would be much more robust in the face of future extensions.

5.5 Linking with code that respects source-language equivalences: Full Abstraction

A different question leads to a different design of compositional compilers: how do we ensure that source-language abstractions are not violated after compilation to a low-level target language? If we preserve source-language abstractions, we end up building compositional compilers that ensure not only the preservation of behavior but also equivalences. Such so-called "fully abstract" compilers compile contextually equivalent source components e_S and e'_S to contextually equivalent target components e_T and e'_T [Ahmed and Blume 2008, 2011; Patrignani et al. 2015] (see Patrignani et al. [2019] for a survey).

An example of this approach is the typed closure conversion pass from a functional language to a target with exceptions shown by New et al. [2016]. Theirs is a type-preserving compiler, which means that a source term e_S of source type τ is compiled to a target term $C^S_T(e_S)$ of translation type τ^+. New et al. show that the compiler is both correct and fully abstract. Their novel proof technique, *universal embedding*, involves back-translation of target contexts into an untyped interpretation of the source, made possible because the source has sums and recursive types and can thus encode the universal type.

As with the other results, in order to show correctness of compilation of components, they need to define some relation ${}_T{\approx}_S$ over components that the compiler will then be shown to satisfy. The particular technique used by this paper is the multi-language technique, so the relation is similar to that in Perconti and Ahmed [2014].

However, since they proved the compiler to be fully abstract, and in particular, exhibited a back-translation function that could translate target terms of translation type $(e_T : \tau^+)$ into source terms $(e_S : \tau)$, the instantiation of the CCC theorem is quite different from that by Perconti and Ahmed [2014] that we showed in §5.3. This demonstrates a powerful information-abstracting principle in the CCC theorem: it should not reveal more than is actually necessary for a user to understand how the theorem can be used. A compiler that has been proved fully abstract requires less trust than one that has not, as we can see in the following:

$\mathcal{L}$	$\{(e_T, \_) \mid$ where e_T is any target component of some translation type $\tau^+\}$
$\widehat{S}$	unchanged source language S
${}_{\widehat{S}}\bowtie_S$	unchanged source linking ${}_S\bowtie_S$
${}_{\widehat{S}}\sqsubset_S$	unchanged source refinement ${}_S\sqsubset_S$
$\Uparrow(\cdot)$	$\Uparrow(e_T, \_) =$ back-translation of e_T to S, written $e_T{}^{\twoheadrightarrow}$

This has some good aspects from both families of approaches we have seen—it allows linking almost as much as the multi-language approach does (only restricting that the target component, at the top level, has a translation type), and retains the simplicity of having $\widehat{S}$ be S.

However, there is a significant restriction that may not be obvious from this summary, which is that the proof of full abstraction relies upon the type translation for the compiler to rule out linking with any behavior that is inexpressible in the source. That, intuitively, is how they are able to define lifting to S and have $\widehat{S}$ be S (unlike Perconti and Ahmed [2014] who can't lift to S since they explicitly wish to allow linking with behavior inexpressible in the source). Concretely, to rule out such linking, [New et al. 2016] use a target language with a modal type system that distinguishes code that can raise uncaught exceptions from code that must have caught all raised exceptions. The output of the compiler has a type that prevents it from being linked with any code that can raise uncaught exceptions.

As before, we must show that the CCC theorem, instantiated with the above, holds based on the work of New et al. [2016]:

$$\forall e_S \in S.\ \forall (e_T, \_) \in \mathcal{L}.\ e_T\ {}_T\bowtie_T\ C_T^S(e_S)\ \ {}_T\sqsubset_S\ \ e_T{}^{\twoheadrightarrow}\ {}_S\bowtie_S\ e_S \tag{1}$$

$$\text{where}\quad (\varnothing_T, \_) \in \mathcal{L} \tag{2}$$

$$\forall e_S.\ (C_T^S(e_S), \_) \in \mathcal{L} \tag{3}$$

$$\Uparrow(\varnothing_T, \_) = \varnothing_S \tag{4}$$

$$\forall e_S.\ \varnothing_S\ {}_S\bowtie_S\ e_S\ \ {}_S\sqsubset_S\ \ e_S \tag{5}$$

$$\begin{array}{l}\forall (e_T, \_) \in \mathcal{L}.\ \forall e_S.\ (\forall c_T.\ c_T\ {}_T\bowtie_T\ e_T\ \ {}_T\sqsubset_T\ \ c_T\ {}_T\bowtie_T\ C_T^S(e_S)) \implies \\ \qquad\qquad (\forall c_S.\ c_S\ {}_S\bowtie_S\ \Uparrow(e_T, \_)\ \ {}_S\sqsubset_S\ \ c_S\ {}_S\bowtie_S\ e_S)\end{array} \tag{6}$$

All side-conditions except for the last one hold trivially. The last holds because we have only included in $\mathcal{L}$ terms that have translation type and thus can be back-translated. The condition then follows from correctness of compilation and correctness of back-translation (Theorem 1.2 and Corollary 5.5 in New et al. [2016]).

The central result follows by combining the back-translation correctness, that shows that $e_T{}^{\twoheadrightarrow} \approx \mathcal{ST}(e_T)$, with correctness of compilation, that $e_S \approx \mathcal{ST}(C_T^S(e_S))$.

Beyond mere correctness, from the approach based on fully abstract compilation we can get truly vertically compositional passes with independent proofs. Equivalence preservation, on its own, does not imply compiler correctness, but a correct and fully abstract compiler like that by New et al. [2016] is proved fully abstract by exhibiting a *back-translation*. A back-translation function maps target components e_T from the linking set $\mathcal{L}$ to source components e_S without looking at the proof φ. If two passes are proved compositionally correct, whatever the method, and they have back-translations, we can define $\Uparrow$ as that back-translation and the linking medium $\widehat{S}$ as the source language S of the respective pass. In that case, we can prove, in general, that the two passes compose to give a two-pass compiler that also satisfies our CCC theorem. In general, equivalence-preserving compilers—whether they preserve contextual equivalence or other notions of equivalence such as noninterference [Bowman and Ahmed 2015], which also requires back-translation—also have useful security properties, so structuring our compilers or target languages to preserve equivalences has other benefits. But we think that the fact compiler-passes with back-translations compose for free is a very useful property, and not something that, to our knowledge, has been observed before. In essence, it gives us vertical compositionality for free. The proof of that composition, which is done in general, is thus agnostic to whatever proof methodology is used to establish the correctness of the individual passes. The theorem statement is the following :

Theorem 5.1 (Passes With Backtranslations Compose).

$$(CCC(C_I^S(\cdot)) \textit{ where } \Uparrow : I \to S) \land (CCC(C_T^I(\cdot)) \textit{ where } \Uparrow : T \to I) \implies CCC(C_T^S(\cdot))$$

Proof. We define the compiler $C_T^S(\cdot)$ as the composition $C_T^I \circ C_I^S$ and the composed lift function $\Uparrow_{ST}$ to be the composition of the two lift functions that exist due to the two passes satisfying CCC. We know we can compose the two lift functions because of the restrictions of their types, which map straight back to source terms and do not use φ terms. All of the conditions follow straightforwardly from the conditions of the two passes, once we define things in this way. We provide mechanized proofs of a generalized version of this theorem in our supplementary materials [Patterson and Ahmed 2019]. □

5.6 Linking with fully specified code: Cito Compiler

We referenced earlier what we term the "fully specified approach" to compositional correctness. This strategy, typified by the Cito language from [Wang et al. 2014], but also similar to what's used in the Cogent language by O'Connor et al. [2016], relies on the module boundaries being given full functional specifications. That is, when compiling a partial program e_S, we already have a full functional specification for every module we will link with. Once this is done, normal semantics preservation, similar to that for whole programs, can be done for this partial program e_S. By analogy, if a function can only be understood as a black box that, given inputs, produces outputs, then it is impossible to specify how it should be compiled without input: this is the essential compositional correctness problem we've been addressing; but if the same function is given an input-output specification, compiling it must simply preserve that specification. While this approach seems to simplify the theorem itself, it complicates the use of the languages since programs must be shown to satisfy functional correctness specifications in order to link, and indeed, we would argue is solving a completely different problem, namely how to do compilation in the context of software verification, rather than how to build verified compilers for unverified software. Given that in order to link against the modules thus compiled the programmer has to prove that their target satisfies the functional correctness specification, the approach only seems viable within the context of a larger verification project, where this kind of activity is already expected.

6 DISCUSSION

6.1 Reflections on Trusting Trust

Trust is, in the final analysis, at the heart of verification. Minimizing the trusted computing base has long been a goal in the security and verification community [Saltzer and Schroeder 1975], with the intended goal of minimizing the amount of code that a user of the system would have to trust to be correct, or perhaps to manually verify. But trust is about more than the code that carries out proof verification, it also involves *understanding* theorems themselves. Just as a user has to trust or manually verify that the unverified code in a verification framework does what they think, they must trust or verify that theorems that make up the formalism of a compiler correctness result actually correspond to the real behavior they expect of the compiler.

What the CCC framework does is help users to isolate the parts that must be trusted, in the form of *parameters* to the framework, as those are the only parts that can attack the validity of the result. Reviewing these parameters for the results in §5, we see that they all require that we trust linking of S components and linking of T components in addition to trusting the formal semantics of S and T as we did for whole-program correctness in §2. But for SepCompCert and for New et al.'s fully abstract compiler, we do not have to trust or understand much more, while Pilsner, the multi-language approach, and Compositional CompCert require that we understand considerably more complex formalism, namely the source-target PILS relation, the multi-language semantics, and interaction semantics, respectively.

The parameters to CCC are the only parts that can invalidate the compiler correctness theorem, but render it invalid they certainly can! For example, consider a CCC theorem defined as such:

$\mathcal{L}$	$\{(e_T, \_) \mid$ any target term $\}$
$\widehat{S}$	unchanged *target* language T
${}_{\widehat{S}}\bowtie_S$	compiles right side and links with left using ${}_T\bowtie_T$
${}_{\widehat{S}}\sqsubset_S$	source-target refinement ${}_T\sqsubset_S$
$\twoheaduparrow(\cdot)$	$\twoheaduparrow(e_T, \_) = e_T$

This could be a perfectly valid CCC theorem, but it does so by taking the target language T to be $\widehat{S}$ and thus cross-language linking first involves *compiling*. A reader of this instantiation should reject this as meaningless, because the definition of the compiler is now part of the parameters, and thus to understand what the theorem says, the reader needs to understand (and trust) the compiler.

And if we require the reader to understand and trust the compiler in order to understand the compiler correctness theorem, there is no point in having a theorem at all! Better to write "please read this compiler and check that it matches your expectations".

While this may seem absurd, in compiler correctness results, this (in more subtle ways) is a very real risk, because if the formalism that a user needs to understand is of similar complexity to the compiler itself, the theorem, however true, is useless! In this sense, there is a very real hard upper limit to how complicated that compiler correctness formalism can be, as if it becomes any more complicated, all hope is indeed lost.

Part of the value of the CCC approach is that it makes very explicit what are the parts that must be understood and what are the details of the proofs that can be ignored. For users, this is clearly beneficial, but is also benefits researchers, as it lets them know how close they are to the danger-zone of "just read the compiler". Trust is a tricky business, but we trust that this can perhaps help.

6.2 Nondeterminism and Lift Functions

We mentioned in §4.1 that defining the lift function can be subtle in the presence of nondeterminism. Consider a compiler from a nondeterministic source to a more deterministic target. In particular, consider a source language with an unspecified order of evaluation of addition operations and a target that only permits addition of variables, eliminating the nondeterminism, which means that the compiler must choose an order of evaluation. For example, if our source component e_S is:

$$\texttt{print(1)} + \texttt{print(2)}$$

then it could have been compiled to the following e_T:

$$\begin{aligned} x_1 &= \texttt{print(1)} \\ x_2 &= \texttt{print(2)} \\ x_1 &+ x_2 \end{aligned}$$

Now, Condition (6) requires, essentially, that lifting e_T will refine e_S. To simplify our discussion, we won't consider what c_S these are linking against, and will consider $\widehat{S}$ to be S. Then, for $\Uparrow(e_T, \varphi)\ {}_S{\sqsubseteq}_S\ e_S$ to hold, every possible behavior of $\Uparrow(e_T, \varphi)$ must be a possible behavior of e_S. We can easily prove the latter in this case, by simply defining the lift function so it does the reverse of the compiler, lifting e_T to e_S.

But now consider the situation where we had instead compiled the following source e'_S, to the same target e_T from above:

$$\begin{aligned} x &= \texttt{print(1)} \\ x &+ \texttt{print(2)} \end{aligned}$$

Now, if we still lift e_T to e_S, then we would not be able to prove the refinement required by Condition (6), i.e., $\Uparrow(e_T, \varphi)\ {}_S{\sqsubseteq}_S\ e'_S$, since one possible behavior of e_S is to print 2 and then 1, but that is not a behavior of e'_S.

To avoid such problems, we need to define the lift function so that it lifts to a source component that is more deterministic than the target but where we haven't added more nondeterminism in than the compiler removed. So in this example, where the compiler is removing all nondeterminism, we can lift to a fully deterministic source, and all the conditions will be satisfied. But it's worth exploring in more detail why these requirements exist.

First, we want to ensure that we lift to a source-like component that has at least as much behavior as the target. This is a consequence of Condition (1): consider an empty e_S, in which case Condition (1) says that $e_T\ {}_T{\sqsubseteq}_{\widehat{S}}\ \Uparrow(e_T, \varphi)$, which requires that *every* behavior of e_T is a possible behavior of $\Uparrow(e_T, \varphi)$. If e_T has more behaviors than $\Uparrow(e_T, \varphi)$, this is violated, so we can't lift to a more deterministic $\widehat{S}$ component.

Second, from Condition (6), we have that if a target component e_T refines a compiled source component $C_T^S(e_S)$, then the lifted version $\Uparrow(e_T, \varphi)$ must refine the source component e_S. In the maximal case, where the behavior of e_T is equal to $C_T^S(e_S)$, this means that the lift function can only introduce behavior up to the point of the behavior of e_S — as the refinement means that all behavior in $\Uparrow(e_T, \varphi)$ must be present in e_S. This, in a sense, gives us an upper bound on what the lift function can do.

Combining these two observations, we have $e_T\ {}_T{\sqsubseteq}_{\widehat{S}}\ \Uparrow(e_T, \varphi)\ {}_{\widehat{S}}{\sqsubseteq}_S\ e_S$, which gives the range of the possibilities for a lift function for a compiler from a nondeterministic source language to a less nondeterministic (or completely deterministic) target language. If we instead had a compiler from a more deterministic source language to a less deterministic target language, the refinements would take place in the other direction, and everything would proceed analogously.

It's not entirely clear that we will always be able to define a lifting to an $\widehat{S}$ component that fulfills these exact requirements. In particular, this requires that the desired lifting be syntactically expressible in the source language—or actually, in $\widehat{S}$, not S. That is, we would need an $\widehat{S}$ in which we can express components that are less deterministic than the target code but not so nondeterministic as to overshoot what the compiler did, which may be difficult, and of course the second part of the restriction depends on the choices made in the compiler, which may indeed be subtle to understand.

It is worth asking if the lift requirements that CCC imposes on compiler correctness results, namely the requirement that every instantiation of CCC be able to provide a lift function that satisfies these conditions is “too strong”. The answer depends on the kind of compiler correctness theorem we are instantiating with. In the case of whole-program and separate-compilation correctness results, if we carefully inspect what Condition (6) requires, then it’s clear that CCC doesn’t impose any requirements stronger than what such results would already prove. This is because in the case of whole-program results, the only element in the linking set is the empty component, and thus Condition (6) follows from Condition (5). Meanwhile, in the case of separate compilers, the requirement that components have been produced by the same compiler means the φ can be the source component that was compiled, and thus the lift function, as it was for SepCompCert, simply chooses the source component that is compiled.

When it comes to *compositional* compiler correctness results, Condition (6) does, of course, impose a meaningful requirement—in particular, when allowing linking with code of very different provenance, getting access to a version of the code that can be linked and has behavior within this range (in terms of determinism) is key—so the requirements of the lift function are not spurious, though they may be not be easy to satisfy. So while one can view the lift requirements as a heavy burden of CCC, as it forces the authors of correctness results to come up with a lift function and prove non-trivial properties about it, these properties are not irrelevant, at least in the most general cases for compositional compilers. Nonetheless, we cannot predict whether future compositional compiler correctness results can come up with clever ways of establishing correctness that might render the lift requirements “too strong”—that is, we have no way of showing that these requirements are necessary as well as sufficient.

6.3 Pedantics about Semantics Preservation

Throughout this pearl, we’ve said that by ${}_{T}\sqsubseteq_{S}$ we mean “semantics preservation”, which we’ve described earlier as “observable behavior is preserved”. But depending on the languages in question, what that means can differ. The simplest case is that both the source and target are deterministic error-free languages. In that case, we expect that there is some set of observations that occur from running programs in the source and target. This could include output printed to the screen, allocations of memory, final results, etc. What exactly these observations are depends on the particular approach to formalization, but should follow from the language, and usually includes programs going wrong, termination, and possibly a final result. From these observations, we can produce a trace of events that occur when the program runs. In the deterministic error-free case, all events that occur in the source trace must appear in the target trace. In many results, the traces should actually be equal, though this may depend on defining the observations to ensure the property. For example, if the source language had large values that, through compilation, ended up being allocated on the heap, the traces for the target should not include heap allocation events if the traces are to be equal.

In the case that either the source or target language is non-deterministic, semantics preservation means a somewhat different thing. If the source is non-deterministic, we usually require only that every trace of the target program is one possible trace of the source. This is known as backwards simulation or sometimes “behavior refinement”, as we require than the target trace *implies* the source trace. Forward simulation is the opposite, that a source trace implies a corresponding target trace. In the case that the target is deterministic, forward simulation implies backwards simulation. The verified compiler writer may also not want to preserve errors in the source language, which is another reason why we may want the target behavior to be a subset of the source behavior. A source program that goes wrong has no meaning, so there is little point in proving that the behavior is preserved.

For most results, the end goal is proving that the target refines the source, i.e., a backwards simulation. For instance, this is the case for [Stewart 2015] and [Neis et al. 2015], while [Perconti and Ahmed 2014] actually prove equivalence since both their source and target are deterministic, type-safe languages. However, it is interesting to note that even results that aim to prove backward simulation might actually prefer to prove forward simulation (which is usually easier to prove) and then show that forward simulation implies backward simulation. For instance, Pilsner uses a mixed simulation that allows forward simulation at deterministic steps (and most of their steps are deterministic) and backward simulation at non-deterministic steps, but they use this to then show the usual notion that target refines source.

6.4 Multi-languages, maligned but still kicking

While CCC captures what it means for a compositional compiler to be correct, it alone is not sufficient for a programmer to actually understand what a program $e_T \bowtie e_S$ means. In particular, without understanding the particular choices of representation that the compiler makes, the programmer does not know how their code e_S is translated to $C^S_T(e_S)$. As a concrete example, we could construct a compiler that translates `true` to `false` and vice-versa. What our theorem would ensure (as would any whole-program correctness theorem) is that the order of branches was also swapped. While the reader might, at first, think that this is a weakness, as we have permitted what seems to be a "wrong" compiler, we note that this is simply an abstraction over data representation. A compiler writer should be free to represent source abstractions in any manner that she chooses, and the form of a generic compiler correctness theorem should not rule that out.

Indeed, we conjecture that given a compositionally correct compiler there exists a multi-language that arises from a source-target *value translation*—which specifies how source-typed (τ) values are converted to target-typed (τ^+) values and vice versa (see Perconti and Ahmed [2014], Patterson et al. [2017])—and source and target operational semantics, where the latter are preserved by the embedding. The value translation, which the user must understand, is not a compiler, as it only translates introduction forms, i.e., a single-argument source function f is translated to a target function $\lambda x.\, \mathcal{TS}(f\ \mathcal{ST}(x))$—essentially an eta-expansion plus language boundaries. The value translation for this multi-language could then be used by the programmer to understand how values flow between the two languages. This would allow them to reason about the meaning of the program $e'_T\ {}_T{\bowtie}_S\ e_S$—defined as $\mathcal{ST}(e'_T)\ {}_{ST}{\bowtie}_{ST}\ e_S$—where within each language the operational semantics of that language hold and the value translation guides crossing between languages.

Our reason for believing that such a multi-language should exist for every compiler comes from the fact that the compiler is compositional. Such a multi-language requires that translating values at any point produces similar results, which for a non-compositional compiler is not guaranteed. It's a research question whether this assertion could be proved, and whether there are additional restrictions on the languages that would be necessary to do so, but we think that it's an interesting avenue of future work.

Note that while we think that such multi-languages should be used for programmer understanding, we do not want to require the CCC theorem to reference them. In particular, this should mean that we never end up with multi-languages with more than two languages. A two-pass, compositionally correct compiler from S to I and I to T should result in a multi-language from the source to the target, where the value translation can be built by composing the two value translations.

7 THE WAY FORWARD

Compiler verification is not, of course, a new problem. But whole program semantics preservation like in Morris [1973] leaves a meaningful distance between the theorem that is proved and the way that compilers are actually used today. Unfortunately, a theorem that does not match reality is not a happy theorem. As the research of Kang et al. [2016] discovered when developing SepCompCert, even verified compilers, proven for whole programs, may contain bugs lurking inside that only appear when partial programs are compiled and linked. And that bug was found in the relatively modest extension to separate compilation; who knows what kind of miscompilations might rear up in the presence of linking with the output of other compilers, or hand-written assembly.

Of course, this pearl is in part a testament to the amount of work that has gone into and continues to go into addressing this problem, and yet the *sheer variety* of solutions that have been proposed hint that perhaps we haven't found the perfect approach yet. But we hope that in accommodating the divergent approaches taken in prior work, we haven't obscured too much our thoughts on the way forward. The reader may wonder, given this experience, "What formalism should I use for my next verification effort?" Some approaches we have shown, like SepCompCert, aim at simple separate compilation, but they do that effectively, and indeed their formalism is now a part of the CompCert compiler as of version 2.7[12]. Others, like the multi-language approach by Perconti and Ahmed [2014], allow flexible linking with arbitrary target code, but require extensive formalism and remain, at this point, theoretical contributions.

The CCC framework exists not only as a way of understanding these disparate approaches, but to aid in the process of reviewing new results. We're happy that more and more, new results come with mechanized proofs, which means that what is critical to review are the theorem statements themselves. And understanding the theorems: their limitations, their expressiveness, and the way they relate to others, is exactly the purpose of our CCC contribution.

We set out with three criteria: to accommodate realistic linking, to account for existing research, and to get a theorem that was not too complex. The first is always hard to assess, but CCC balances simplicity and an ability to account for the differences in various prior approaches, and the variety of these approaches gives us hope that future efforts may fit in as well.

We've also demonstrated that for compilers with back-translations, normally true of equivalence-preserving compilers, the important and often difficult process of proving vertical compositionality can be proven abstractly. This is obviously appealing, as it means that in order to construct verified multi-pass compilers, not only is it trivial to prove that the passes compose, but the proofs of each pass need not even rely upon the same formalism, as the proof of the vertical compositionality lemma relies solely on the CCC theorem. This is in contrast to any of the other approaches described in this pearl, and truly something that we believe researchers should strive for.

We believe that rather than focusing on trying to invent a single best technique for verification, future compositional efforts should aim to support passes that can be composed as flexibly as possible, to hopefully support future extensions without requiring redundant re-verification. They should also support linking with code that did not originate in the same source language, since often, excluding separate compilation, the reason why we are linking is to gain access to some feature that did not or could not originate in the source language. In short, we want verified compilers that support both true Vertical Compositionality and Source-Independent Linking!

Though we've suggested that fully abstract compilers may be one possible way to get good vertical compositionality, they come with a weakness, beyond being difficult to prove: they generally only permit linking with code that extensionally behaves like some code expressible in the source language. Technically, a fully abstract compiler can never allow linking with any code that can disturb equivalences from the source language. This rules out common cases of linking with libraries written in lower-level languages to gain access to features that do not exist in a simple high-level language, for example, adding threading primitives to a language without them ([Ahmed

[12]See http://compcert.inria.fr/release/Changelog

2015; Mates et al. 2019]). There is some work arguing that rather than a flaw of the fully abstract compilation approach, this is instead a problem of incomplete source-language specifications that fail to account for interoperability with features inexpressible in that language [Patterson and Ahmed 2017]. The idea is that when linking, programmers should be able to annotate their programs to reflect the kind of linking behavior that they expect, even linking with features unavailable in the source, and then fully abstract compilers would ensure linking respects those annotations. Whether that idea or others work out, there is a lot of exciting work to be done in this area to get both vertical compositionality and source-independent linking.

The key challenge with compiler verification, is not whether the proof is true, but rather, does the compiler in the proof correspond to the compiler in reality? Unquestionably, the work on whole-program compiler correctness over the past several decades has been incredibly important. But as we move forward, we must ask more from our theorems, and we should present them in the simplest way possible so that users of our compilers can be sure that the way that they are using the compiler corresponds to what was indeed verified. Perhaps someday, there will be a single approach that will eliminate the need for the type of unifying framework that we have described in this pearl. But until then, we hope that CCC will help to serve as a navigational aid for the landscape of compiler verification efforts, to both illuminate for readers and reviewers, and set out guidelines for researchers, and make all of this work a little easier to understand, and thus benefit from.

ACKNOWLEDGMENTS

We thank the ICFP'19 reviewers for their valuable feedback and reviewers from POPL'18 for helpful feedback on an earlier version of this work. We also thank Lennart Beringer for an enlightening conversation at PLDI'17 that helped us better understand Compositional CompCert. This material is based on research supported by the National Science Foundation under grants CCF-1816837, CCF-1618732, CCF-1453796, and CCF-1422133. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

A CCC WITH MANY MODULES

Our definition can be generalized to handle an arbitrary number of target components that are linked with the compiled source component. This changes both the primary definition and the last side condition, as follows. This is the definition we used when proving the corollaries in Coq, provided in our supplementary materials [Patterson and Ahmed 2019].

Theorem A.1 (CCC: Compositional Compiler Correctness).

$$\exists \Uparrow.\ \forall e_S \in S.\ \forall \overline{(e_T, \varphi)} \in \mathcal{L}.\ \overline{e_T}\ {}_T{\bowtie}_T\ C_T^S(e_S)\ {}_T{\sqsubseteq}_{\widehat{S}}\ \overline{\Uparrow(e_T, \varphi)}\ {}_{\widehat{S}}{\bowtie}_S\ e_S \tag{1}$$

Notation: We write $\overline{e}$ *as shorthand for* $e_1, \ldots, e_n$*, and* $e_1, \ldots, e_n\ {}_{L_1}{\bowtie}_{L_2}\ e'_1, \ldots, e'_m$ *means interleaving listed components in some fixed way, where the interleaving is fixed for each instantiation of the theorem.*

$$\text{where}\quad (\varnothing_T, \varphi_\varnothing) \in \mathcal{L} \tag{2}$$

$$\forall e_S.\ \exists \varphi.\ (C_T^S(e_S), \varphi) \in \mathcal{L} \tag{3}$$

$$\Uparrow(\varnothing_T, \varphi_\varnothing) = \varnothing_{\widehat{S}} \tag{4}$$

$$\forall e_S.\ \varnothing_{\widehat{S}}\ {}_{\widehat{S}}{\bowtie}_S\ e_S\ {}_{\widehat{S}}{\sqsubseteq}_S\ e_S \tag{5}$$

$$\begin{aligned} \forall \overline{(e_T, \varphi)} \in \mathcal{L}.\ \forall \overline{e_S}.\ (\forall c_T.\ c_T\ {}_T{\bowtie}_T\ \overline{e_T}\ {}_T{\sqsubseteq}_T\ c_T\ {}_T{\bowtie}_T\ \overline{C_T^S(e_S)}) \implies \\ (\forall c_S.\ c_S\ {}_{\widehat{S}}{\bowtie}_S\ \overline{\Uparrow(e_T, \varphi)}\ {}_{\widehat{S}}{\sqsubseteq}_S\ c_S\ {}_S{\bowtie}_S\ \overline{e_S}) \end{aligned} \tag{6}$$

REFERENCES

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *European Symposium on Programming (ESOP)*. 69–83.

Amal Ahmed. 2015. Verified Compilers for a Multi-Language World. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. 15–31.

Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *International Conference on Functional Programming (ICFP), Victoria, British Columbia, Canada*. 157–168.

Amal Ahmed and Matthias Blume. 2011. An Equivalence-Preserving CPS Translation via Multi-Language Semantics. In *International Conference on Functional Programming (ICFP), Tokyo, Japan*. 431–444.

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-Dependent Representation Independence. In *ACM Symposium on Principles of Programming Languages (POPL), Savannah, Georgia*.

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, Step-Indexing and Compiler Correctness. In *International Conference on Functional Programming (ICFP), Edinburgh, Scotland*.

Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *European Symposium on Programming (ESOP)*.

William J. Bowman and Amal Ahmed. 2015. Noninterference for Free. In *International Conference on Functional Programming (ICFP), Vancouver, British Columbia, Canada*.

Adam Chlipala. 2007. A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, California*.

Matthias Felleisen. 1990. On the Expressive Power of Programming Languages. In *Science of Computer Programming*. Springer-Verlag, 134–151.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*. 595–608.

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke logical relation between ML and assembly. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*.

Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards certified separate compilation for concurrent programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 111–125.

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*. ACM, 178–190.

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*.

Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.

Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under Control: Compositionally Correct Closure Conversion with Mutable State. In *ACM Conference on Principles and Practice of Declarative Programming (PPDP)*.

Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *ACM Symposium on Principles of Programming Languages (POPL), Nice, France*. 3–10.

John McCarthy. 1959. A Basis for a Mathematical Theory of Computation. *Studies in Logic and the Foundations of Mathematics*, 33–70.

John McCarthy and James Painter. 1967. Correctness of a compiler for arithmetic expressions. American Mathematical Society, 33–41.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*. 271–283.

F Lockwood Morris. 1973. Advice on structuring compilers and proving them correct. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 144–152.

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language. In *International Conference on Functional Programming (ICFP), Vancouver, British Columbia, Canada*.

Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation via Universal Embedding. In *International Conference on Functional Programming (ICFP), Nara, Japan*.

Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. 2016. Refinement through restraint: Bringing down the cost of verification. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 89–102.

Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Transactions on Programming Languages and Systems* 37, 2, Article 6 (April 2015), 50 pages.

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *Comput. Surveys* 51, 6, Article 125 (Feb. 2019), 36 pages.

Daniel Patterson and Amal Ahmed. 2017. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 12:1–12:15. https://doi.org/10.4230/LIPIcs.SNAPL.2017.12

Daniel Patterson and Amal Ahmed. 2019. CCC: Supplementary Materials. https://dbp.io/pubs/2019/ccc/

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: Reasonably Mixing a Functional Language with Assembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Barcelona, Spain*.

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-Language Semantics. In *European Symposium on Programming (ESOP)*.

Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Jérémie Koenig, and Yuchen Fu. 2015. A Compositional Semantics for Verified Separate Compilation and Linking. In *Proceedings of the 2015 Conference on Certified Programs and Proofs (CPP '15)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2676724.2693167

Jerome H. Saltzer and Michael D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (September 1975), 1278–1308. http://web.mit.edu/Saltzer/www/publications/protection/index.html

Jaroslav Ŝevčik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-memory concurrency and verified compilation. *ACM SIGPLAN Notices* 46, 1 (2011), 43–54.

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*.

James Gordon Stewart. 2015. *Verified Separate Compilation for C*. Ph.D. Dissertation. Princeton University.

Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler Verification Meets Cross-Language Linking via Data Abstraction. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*.

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An abstract stack based approach to verified compositional compilation to machine code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 62.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Jose, California*.