Verified Compilers for a Multi-Language World*

Amal Ahmed

Northeastern University
amal@ccs.neu.edu

Abstract

Though there has been remarkable progress on formally verified compilers in recent years, most of these compilers suffer from a serious limitation: they are proved correct under the assumption that they will only be used to compile *whole* programs. This is an unrealistic assumption since most software systems today are comprised of components written in different languages – both typed and untyped – compiled by different compilers to a common target, as well as low-level libraries that may be handwritten in the target language.

We are pursuing a new methodology for building verified compilers for today's world of multi-language software. The project has two central themes, both of which stem from a view of *compiler correctness as a language interoperability problem*. First, to specify correctness of component compilation, we require that if a source component s compiles to target component t, then t linked with some arbitrary target code t' should behave the same as s interoperating with t'. The latter demands a formal semantics of interoperability between the source and target languages. Second, to enable safe interoperability between components compiled from languages as different as ML, Rust, Python, and C, we plan to design a gradually type-safe target language based on LLVM that supports safe interoperability between more precisely typed, less precisely typed, and type-unsafe components. Our approach opens up a new avenue for exploring sensible language interoperability while also tackling compiler correctness.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages, D.3.1 Formal Definitions and Theory, D.3.4 Processors

Keywords and phrases verified compilation, compositional compiler correctness, multi-language semantics, typed low-level languages, gradual typing

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.15

1 Landscape of Verified Compilation

The field of compiler verification has witnessed considerable progress in the last decade following the pioneering work on CompCert [29, 30]. The latter uses the Coq proof assistant to both implement and verify a multi-pass optimizing compiler from C to assembly, proving that the compiler preserves semantics of source programs. Several other compiler verification efforts have successfully followed CompCert's lead and basic methodology to verify increasingly sophisticated compilers for increasingly realistic languages, for instance, focusing on just-in-time compilation [37], multithreaded Java [31], C with relaxed memory concurrency [48], LLVM passes [62], and imperative functional languages [15, 28].

Unfortunately, all of the above projects prove correctness assuming that the compiler will only be used to compile whole programs. But this assumption contradicts the reality of how we use these compilers. The whole programs that we actually run are almost never the output of a single compiler: they are composed by linking code from various places, including

* This work is supported by the National Science Foundation and a Google Faculty Research Award.

1st Summit on Advances in Programming Languages (SNAPL'15).
Eds.: Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett; pp. 15–31

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the runtime system, libraries, and foreign functions, all potentially compiled using different compilers and written in different languages (e.g., Java, Python, C, and even handcrafted assembly). For today's world of multi-language software, we need verified compilers that guarantee correct compilation of *components*.

Formally verifying that components are compiled correctly – often referred to as *compositional compiler correctness* – is a difficult problem. A key challenge is how to state the compiler correctness theorem for this setting. CompCert's compiler correctness theorem is easy to state thanks to the whole program assumption: informally, it says that if a source program P_S compiles to a target program P_T, then running P_S and P_T results in the same trace of observable events. The same sort of theorem does not make sense when we compile a component e_S to a component e_T: we cannot "run" a component since it is not a complete program.

Part of the challenge is that any correct-component-compilation theorem should satisfy two important properties: (1) it should allow compiled components to be linked with target components of arbitrary provenance, including those compiled from other languages (dubbed *horizontal compositionality* in the literature); and (2) it should support verification of multi-pass compilers by composing proofs of correctness for individual passes (dubbed *vertical compositionality*). These are nontrivial requirements. There have been several notable efforts at compositional compiler verification in recent years but none offer a proof methodology that fully addresses the dual challenges of horizontal and vertical compositionality. The earliest work, by Benton and Hur [12, 13, 26], specifies compiler correctness in a way that does not scale to multi-pass compilers. Both that work and the recent work on Pilsner [41] only supports linking with separately compiled components from the *same* source language. The recent work on Compositional CompCert [51] supports linking across the languages in CompCert's compilation pipeline, but all of these share the same memory model; it is not clear how to extend the approach to support linking across languages with different memory models. (We discuss related work in detail in §3.2.)

Let us look at why compiler correctness becomes challenging in the context of multi-language software. The issue is that real software systems often link together components from multiple languages with *different expressive power* and *different guarantees*. When compiling source language S, what does it mean for the compiler to "preserve the semantics of source code" when a compiled component e_T may be linked with some e'_T from a language S' that is *more expressive* or provides *fewer guarantees* as compared to S?

- Suppose S does not support first-class continuations (call/cc) but S' does. For instance, e'_T, compiled from language S', might be a continuation-based web server that provides call/cc-based primitives that a web programmer can use for creating client-server interaction points [46, 27, 47, 42]. The programmer, meanwhile, might use language S to develop her web application – making use of the afore-mentioned primitives which allow her to write her program in a more direct style [46, 27] – and compile it to e_T. When we link these components and run, e'_T disrupts the control flow of e_T in a way that is useful and desirable but does not mirror any whole-program behavior in the source language S. How, then, do we specify that the compiler "preserves semantics of source code"?
- Suppose S supports strong static typing (e.g., ML) while S' is dynamically typed (e.g., Scheme) or weakly typed (e.g., C). Again, linking e'_T with e_T might result in code that does not mirror any whole-program behavior in the source language S. For instance, (1) the C component may try to access memory it has already freed; (2) the C component may write past the end of a C array, but in doing so overwrite a memory location owned by an ML module; or (3) the Scheme component may apply an ML function that expects

an argument of type $\forall\alpha.\alpha \to \alpha$ to the boolean negation function, thus violating ML's guarantee that a function of this type will behave parametrically with respect to its input. The first interaction (with C) seems entirely reasonable since the error is in C code and does not affect the behavior of the ML component. The second interaction (with C) is clearly unacceptable as it can violate ML's type safety guarantee. The third interaction (with Scheme) may or may not be considered reasonable: it does not violate ML's type safety guarantee, but it does violate ML's parametricity guarantee. In what sense, then, can we say that the compiler "preserves semantics of source code"?

The above examples suggest that we need a novel way of understanding and specifying what we mean by compiler correctness in the context of multi-language software.

2 Our Approach and Research Goals

My research group at Northeastern is working on techniques for building realistic, multi-pass *verified compositional compilers* that guarantee correct compilation of components and formally support linking with arbitrary target code, no matter its source. We follow the tenet that ***compositional compiler correctness is a language interoperability problem***. We are in the early stages of a project that will require years of investigation and evaluation. Our long-term vision, depicted in Figure 1, is to have verified compositional compilers from languages as different as C, Python, ML, Rust, Coq's Gallina, and Hoare Type Theory (HTT) [38, 40, 39] to a common low-level target language that supports safe interoperability between more precisely typed, less precisely typed, and type-unsafe components. Our current focus is on laying the groundwork for realizing this vision through two central contributions:

1. Development of a proof methodology for verifying compositional compilers. The defining feature of our approach is that the specification of compositional compiler correctness relies on a formal semantics of interoperability between source components and target code using multi-language semantics in the style of Matthews and Findler [33].(We have already done extensive work on developing this methodology [45]; we give further details in §3.) To demonstrate the viability of this approach, we plan to verify compositional type-preserving compilers for subsets of ML and Rust.
2. Design of a *gradually type-safe* LLVM-like target language (dubbed GTVM, see Figure 1) that supports safe interoperability between components that are statically type-safe (e.g., produced by our type-preserving ML and Rust compilers), dynamically type-safe (e.g., compiled from Python or Scheme), or potentially unsafe (e.g., compiled from C). We plan to build on Vellvm (verified LLVM) [57, 61, 62] by first developing TTVM (*tightly typed* LLVM), a statically type-safe version of the LLVM IR formalized in Vellvm, and then design GTVM as a gradually typed extension of TTVM. GTVM will make use of casts (think: coercions or contracts) to ensure safe interoperability between the more precisely typed, less precisely typed, and type-unsafe parts of the language – the latter unsafe part is just standard LLVM IR which has types but is not type-safe. We will compile GTVM to the LLVM IR, inserting wrappers that perform dynamic checks to ensure safe coercion. Thus, compilers targeting our gradually type-safe LLVM can continue to leverage the optimizations provided by the LLVM compiler infrastructure [54] or the verified optimizations provided by Vellvm [57, 62].

Specifying compositional compiler correctness for a multi-language world. Informally, if a component e_S compiles to a component e_T then compiler correctness should require that

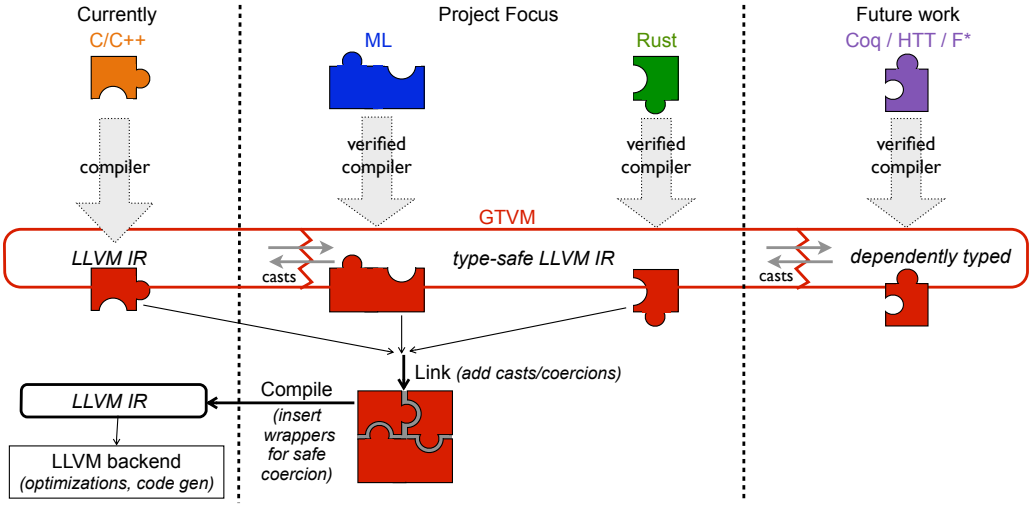

Figure 1 Research planned as part of this project and potential future work.

e_S is "equivalent" to e_T. But how can we *formalize* this notion of "equivalence" between source and target components? Observe that to *use* a compiled component e_T, we will link it with some other target-level component e'_T to obtain a whole program that can be run. Intuitively, therefore, compiler correctness should guarantee that the operational behavior of this resulting target program is the same as the operational behavior of e_S linked with e'_T. Therefore, to formally state that "a component is compiled correctly," we need to formalize the semantics of interoperability between source and target code. For a multi-pass compiler we propose to do this in a modular fashion. For instance, if the compiler consists of two passes, from source language S to intermediate language I to target language T, we define a combined language SIT that embeds these three languages and formalizes the semantics of interoperability between each pair of adjacent languages using boundaries in the style of Matthews and Findler's multi-language semantics [33]. We can stack these boundaries to allow interoperability between the source and target of the compiler, e.g., $\mathcal{SI}(\mathcal{IT}(e_T))$, which we abbreviate to $\mathcal{SIT}(e_T)$, allows a target component e_T to be used from within an S-language expression. Compiler correctness can now be stated as observational equivalence in the combined language: if e_S compiles to e_T, then e_S is observationally equivalent to $\mathcal{SIT}(e_T)$. Direct proofs of observational equivalence – also known as contextual equivalence – are known to be intractable. Therefore, we define a *logical relation* for the combined language that corresponds to contextual equivalence and use that to carry out the proof of compiler correctness. Note that we do not use the multi-language semantics for running actual multi-language programs; its purpose is to serve as a *specification* of the desired source-target relationship, allowing us to state and prove compiler correctness. This specification also enables reasoning about the whole-program behavior of e_T linked with e'_T in terms of the whole-program behavior of e_S linked with $\mathcal{SIT}(e'_T)$. Most importantly, note that we have not imposed any restrictions on the provenance of e'_T. We give further details in §3.

Why ML and Rust? All of the existing work on compositional compiler correctness has either (a) focused on unsafe C-like source languages [51, 59] or (b) assumed that code produced by a verified compiler from a type-safe language will only be linked with code produced by another verified compiler from the *same* source language [12, 13, 26, 41]. We instead focus on compiling the statically typed languages ML and Rust while allowing linking with code

of arbitrary provenance. This is a real-world scenario with interesting semantic challenges for interoperability – specifically, how to maximize interoperability with less precisely typed and type-unsafe components while ensuring that those interactions respect the ML or Rust type system. We believe that these interoperability challenges make compositional compiler correctness harder to establish for these languages. Languages that provide fewer guarantees (e.g., C) are less picky in terms of what they can interoperate with, which makes it less semantically challenging to ensure correct compositional compilation.

Rust is a systems programming language with type-system support for safe memory management. In essence, it supports *affine* types which indicate that a resource – in this case, memory cells – can be used at most once. Since ML does not support affine types, we will have to ensure that ML respects Rust's affine typing guarantees.

GTVM: a gradually type-safe LLVM IR. As the above discussion suggests, to prove compiler correctness in the context of multi-language software, we must specify a formal semantics of interoperability (1) between source and target code, and (2) between more precisely typed, less precisely typed and type-unsafe code. For instance, for our ML compiler, we will have to specify how an ML source component e_S interoperates with a target component e'_T that may have been compiled from Scheme or C. We believe that safe interoperability between more and less typed and type-unsafe should be investigated at the level of the target language so that the benefits of this effort can be reaped by all compilers that target the language. That is the philosophy underlying our investigation of GTVM.

To understand the design of GTVM, it is useful to think of it in two layers: (1) a statically type-safe core, TTVM (tightly typed LLVM), and (2) a gradual typing extension, GTVM, that extends TTVM with dynamically type-safe code of type `dyn` and unsafe, standard LLVM IR code of type `un`. Our goal is a gradual type system that ensures that the typing (and parametricity) guarantees provided by more precise types cannot be violated by the less precisely typed parts of the language.

The statically type-safe core language, TTVM, should provide a rich enough type system to adequately serve as a target for type-preserving compilers from different statically typed languages. Since we wish to compile ML and Rust, our TTVM will need to support at least type abstraction and affine types in addition to the standard LLVM types. Even within this TTVM core, we will need a system of casts (contracts) to mediate between more precise and less precise types to support interoperability between code compiled from different source languages. For instance, we wish to allow code compiled from ML and Rust to interoperate, which means we will need to design casts (probably along the lines of Tov and Pucella [55]) that protect an affinely typed resource (from Rust) from being used more than once, even if it is passed to a function (from ML) that knows nothing about affine types and might freely duplicate the resource.

GTVM will let compiler writers choose whether to target the statically type-safe, dynamically type-safe, or unsafe parts of the language (or some mix of the three) depending on the nature of their source language, and *depending on how restrictive a form of interoperability they want.* ***The lever that provides control over interoperability is the compiler's type translation.*** For instance, when compiling ML, picking the most informative type translation (relative to source types) will guarantee that interoperability with other languages respects the source type system – including parametricity guarantees, as long as the compiler doesn't monomorphize. At the other extreme, translating all ML code to the type `un` says that interoperability with other code need not respect ML's type system guarantees.

Our longer-term goal is to enrich the statically typed core of GTVM with dependent

types in the style of Hoare Type Theory (HTT) [38, 40]. HTT incorporates specifications – in the style of Hoare logic or separation logic – into types and makes it possible to formally specify and reason about effects. A type system based on HTT would allow us to express rich invariants about memory layout, separation, and resource usage, which will be important for specifying low-level conventions that affect interoperability at the LLVM level.

Enabling secure compilation via target-level types. Compiler correctness is about preservation of dynamic semantics, but we are interested in an architecture that can also support the development of secure (or fully abstract) compilers. Informally, secure compilation guarantees that compiled components will remain as secure as their source-level counterparts when executed within target contexts of arbitrary origin. More formally, a fully abstract compiler guarantees that if two source components have the same observable behavior in all well-typed source contexts then their compiled versions must have the same observable behavior in all appropriately-typed target contexts. In prior work [4], we studied how compilers can be made fully abstract by changing the compiler's type translation to ensure that compiled code is never linked with target contexts whose behavior does not match that of some source context. By equipping GTVM with an expressive type system, we wish to offer compiler writers the facility to pick different degrees of *protection* of compiled components from their target contexts – ranging from no protection at all (when compiled code has type `un` and the verified compiler only guarantees preservation of dynamic semantics), all the way through fully abstract compilation – via their choice of type translation.

The rest of this paper explains our approach to specifying compiler correctness (§3) and then outlines our research plans and anticipated challenges (§4).

3 Proof Methodology for Verified Compilation of Components

In recent work [45], we have demonstrated the viability of our multi-language-semantics approach, using it to prove correctness of a two-pass compiler that performs closure conversion and heap allocation, translating a polymorphic source language with recursive types to a target that also features dynamically allocated mutable memory. In particular, we support linking of compiled code with code that performs state effects that cannot be expressed in the source. This work was the first multi-pass, compositional compiler-correctness result. We believe that it is, to date, the only approach that supports linking with code that cannot be expressed in the verified compiler's own source language. We plan to use this methodology to build verified compositional compilers for ML and Rust. Below, we explain our approach in more detail, compare it to related work, and illustrate our methodology using typed closure conversion [34, 36, 3] as a case study.

3.1 Specifying compiler correctness using multi-language semantics

The compiler correctness theorem should say that if a component e_S compiles to e_T, then some desired relationship $e_S \simeq e_T$ holds between e_S and e_T – intuitively, they should "behave the same." But how do we formally specify $e_S \simeq e_T$? To answer this question, consider how the compiled component is actually used: it needs to be linked with some e'_T, creating a whole program that can be run. This e'_T may have been handwritten in the target language or produced by another compiler, possibly from a different source language. Of course, it doesn't make sense to link with *any* e'_T: at the very least, e'_T should adhere to the same calling conventions that e_T does. Moreover, since our target language is typed – with types

`dyn` and `un` in additional to the more standard types – that means that e_T can only be linked with components of a certain type because we want the resulting whole program to be well typed. Informally, then, the compiler correctness theorem should guarantee that if we link e_T with an appropriately typed e'_T then the resulting target-level program should correspond to the source component e_S linked with e'_T. Formally speaking, what does it mean to "link a source component with a target component" and what are the rules for running the resulting source-target hybrid? We have argued that these questions demand a *semantics of interoperability* between the source and target languages. Next, we explain how to specify such semantics.

Consider a two-pass compiler from a source language S (in blue) to intermediate language I (in red) to target language T (in purple). The first pass translates S components $\mathsf{e_S}$ of type τ to I components $\mathrm{e_I}$ of type $\tau^{\mathcal{I}}$, where $\tau^{\mathcal{I}}$ denotes the type translation of τ. As is usual with type-preserving compilation, the type $\tau^{\mathcal{I}}$ provides a simple means of expressing any compiler invariants about representation and/or layout of the transformed term $\mathrm{e_I}$ that are useful to keep track of as we compile. The second pass analogously translates I components $\mathrm{e_I}$ of type τ to T components $\mathsf{e_T}$ of type $\tau^{\mathcal{T}}$, where $\tau^{\mathcal{T}}$ is the type translation of τ.

To define the semantics of interoperability between these languages, we *embed* them all into one language, *SIT*, and add syntactic boundary forms between each pair of adjacent languages in the style of Matthews and Findler [33] and our own prior work [4, 45]. For instance, the term $\mathcal{IS}^{\tau}(\mathsf{e_S})$ allows an S component $\mathsf{e_S}$ of type τ to be used as an I component of type $\tau^{\mathcal{I}}$, while ${}^{\tau}\mathcal{SI}(\mathrm{e_I})$ allows an I component $\mathrm{e_I}$ of translation type $\tau^{\mathcal{I}}$ to be used as an S component of type τ. Similarly, we have boundary forms $\mathcal{TI}$ and $\mathcal{IT}$ for the next language pair. Non-adjacent languages can interact by stacking up boundaries: for example, $\mathcal{SI}(\mathcal{IT}\,\mathsf{e_T})$ (abbreviated $\mathcal{SIT}(\mathsf{e_T})$) allows a T component $\mathsf{e_T}$ to be embedded in an S term.

Design principles for multi-language system. Our goal is for the *SIT* interoperability semantics to give us a useful specification of when a component in one of the embedded languages should be considered equivalent to a component in another language. But how do we know if that specification is correct? There are three essential properties that the combined language must satisfy.

First, the operational semantics of *SIT* must be designed so that the original languages are embedded into *SIT* unchanged: running an *SIT* program that's written solely in one of the embedded languages is identical to running it in that language alone. For instance, execution of the T program $\mathsf{e_T}$ proceeds in exactly the same way whether we use the operational semantics of T or the augmented semantics for *SIT*. Second, the typing rules must be similarly embedded: a component that contains syntax from only one underlying language should typecheck under that language's individual type system if and only if it typechecks under *SIT*'s type system. The final property we need is *boundary cancellation* which says that wrapping two opposite language boundaries around a component yields the same behavior as the underlying component with no boundaries: for example, any $\mathsf{e_S} : \tau$ must be contextually equivalent to ${}^{\tau}\mathcal{SI}(\mathcal{IS}^{\tau}\mathsf{e_S})$, and any $\mathrm{e_I} : \tau^{\mathcal{I}}$ must be equivalent to $\mathcal{IS}^{\tau}({}^{\tau}\mathcal{SI}\mathrm{e_I})$. Note that two components e_1 and e_2 are considered contextually equivalent in language *SIT* (written $e_1 \approx^{ctx}_{SIT} e_2$) if there is no well-typed *SIT* program context that can tell them apart.

Compiler correctness. We state the correctness criterion for our compiler as a contextual equivalence: if $\mathsf{e_S} : \tau$ compiles to $\mathrm{e_I}$, then $\mathsf{e_S} \simeq \mathrm{e_I}$, where: $\mathsf{e_S} \simeq \mathrm{e_I} \stackrel{\text{def}}{=} \mathsf{e_S} \approx^{ctx}_{SIT} {}^{\tau}\mathcal{SI}(\mathrm{e_I}) : \tau$ and similarly for the next pass:

If $\mathrm{e_I} : \tau$ compiles to $\mathsf{e_T}$, then $\mathrm{e_I} \simeq \mathsf{e_T}$, where: $\mathrm{e_I} \simeq \mathsf{e_T} \stackrel{\text{def}}{=} \mathrm{e_I} \approx^{ctx}_{SIT} {}^{\tau}\mathcal{IT}(\mathsf{e_T}) : \tau$.

Since contextual equivalence is transitive, our framework achieves vertical compositionality immediately: it is easy to combine the two correctness proofs for the individual compiler passes, to get the correctness result for the entire compiler:

If $\mathsf{e_S}$ compiles to $\mathsf{e_T}$, then $\mathsf{e_S} \approx^{ctx}_{SIT} {}^{\tau}\mathcal{SIT}(\mathsf{e_T}) : \tau$.

All of the above properties are stated as contextual equivalences but direct proofs of contextual equivalence are usually intractable. We use the standard technique of defining a logical relation for the combined language SIT that is sound and complete with respect to contextual equivalence. Defining the logical relation becomes more challenging as the demands of interoperability increase: e.g., when all the interoperating languages support type abstraction. (See the paper [45] for details.)

Reasoning about linking. Our approach enjoys a strong horizontal compositionality property: we can link with any target component $\mathsf{e'_T}$ that has an appropriate type, with no requirement that $\mathsf{e'_T}$ was produced by any particular means or from any particular source language. Specifically, if $\mathsf{e_S} : \tau' \to \tau$ expects to be linked with a component of type τ' and compiles to $\mathsf{e_T}$, then $\mathsf{e_T}$ will expect to be linked with a component of type $((\tau')^{\mathcal{I}})^{\mathcal{T}}$. If $\mathsf{e'_T}$ has this type, then using our compiler correctness theorem, we can conclude that: $(\mathsf{e_S} {}^{\tau'}\mathcal{SIT}(\mathsf{e'_T})) \approx^{ctx}_{SIT} \mathcal{SIT}(\mathsf{e_T}\ \mathsf{e'_T}) : \tau$.

3.2 Comparison with related work

The literature on compiler verification spans almost five decades but is mostly limited to whole-program compilation. We refer the reader to the bibliography by Dave [16] for compilation of first-order languages, and to Chlipala [15] for compilation of higher-order functional languages. Here we discuss only recent work on compositional compiler correctness.

The approach advocated by Benton and Hur [12, 13] involves formalizing correct compilation of components using a logical relation that specifies when a source term e_S semantically approximates target code e_T and vice versa (written $e_S \simeq e_T$). Using this approach, they verifed a compiler from STLC with recursion (and later from System F) to an SECD machine, proving that if source component e_S compiles to target code e_T, then e_S and e_T are logically related. Later, Hur and Dreyer [26] used essentially the same strategy to verify a compiler from an idealized ML to assembly. However, this strategy of setting up a logical relation between source and target has two serious drawbacks. First, the approach does not scale to multi-pass compilers because the source-target logical relations defined for each pass do not compose. Second, if we compile an S component e_S to e_T and then wish to link e_T with some arbitrary e'_T, the only way to check if it's okay to link with e'_T (i.e., if the compiler correctness theorem allows it) is to *come up with a source-level component* e'_S and show that $e'_S \simeq e'_T$. It may be possible to come up with e'_S when e'_T is a few lines long, but it would be infeasible when e'_T consists of hundreds of lines of assembly. Worse, the question of whether such an e'_S exists in language S is undecidable: for instance, there is no e'_S in the case of the continuation-based web server example discussed in §1. By comparison, our approach simply requires type-checking e'_T to ensure that it can be sensibly linked with t.

Neis *et al.* [41] recently developed Pilsner, a verified separate compiler from a typed, higher-order imperative language to an untyped target. Pilsner also suffers from the second drawback discussed above for the Benton-Hur and Hur-Dreyer work – put another way, Pilsner assumes that compiled code will only be linked with code compiled by a (possibly different) verified compiler from the *same source language*. Instead of a logical relation between source and target code, Pilsner uses parametric inter-language simulations (PILS)

between the source and target of each compiler pass. Unlike Kripke logical relations, PILS do compose transitively and can be used to verify a multi-pass compiler.

Stewart *et al.* [51] recently reported on Compositional CompCert, a verified separate compiler for C. (This generalizes their prior work [14] which allowed shared-memory system calls from C, but could not accommodate mutually recursive inter-module dependencies.) Whereas we define a multi-language semantics for interoperability between source and target, Stewart *et al.* have devised an *interaction semantics*, a protocol-oriented operational semantics that accommodates interoperation between different languages. They have shown that CompCert's intermediate languages – all of which share the same C-like memory model – can be "plugged into" this interaction semantics. It is not clear how to extend interaction semantics to support interoperability between C and high-level, strongly typed languages like ML, or more generally, to accommodate compilers whose source and target languages use different memory models.

3.3 Our proof methodology applied to closure conversion

As an example, we show how our methodology can be used to prove compositional correctness of typed closure conversion [34, 36] for the simply-typed lambda calculus (STLC). (We elide many details; see [45].)

Step 0: Specify the source language. The source language S is call-by-value STLC with booleans. The syntax of the language is as follows:

$$
\begin{array}{llll}
\textit{Types} & \sigma & ::= & \mathsf{bool} \mid \sigma_1 \to \sigma_2 \\
\textit{Values} & \mathsf{v} & ::= & \mathsf{x} \mid \mathsf{true} \mid \mathsf{false} \mid \lambda \mathsf{x} : \sigma.\, \mathsf{e} \\
\textit{Expressions} & \mathsf{e} & ::= & \mathsf{v} \mid \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 \mid \mathsf{e}_1\ \mathsf{e}_2
\end{array}
$$

Step 1. Pick appropriate target language and define translation. Closure conversion is a compiler transformation that collects a function's free variables in a tuple called a *closure environment* that is passed as an additional argument to the function, thus turning the function into a closed term. The closed function is paired with its environment to create a *closure*. The basic idea of *typed* closure conversion goes back to Minamide *et al.* [34], whom we follow in using an existential type to abstract the type of the environment. This ensures that two functions with the same type, but different free variables still have the same type after closure conversion: the abstract type hides the fact that the closures' environments have different types. Thus our target language for closure conversion must support existential types as well as tuples. It also supports multi-argument functions. We define the target language T as follows:

$$
\begin{array}{llll}
\textit{Types} & \boldsymbol{\tau} & ::= & \mathbf{bool} \mid (\overline{\boldsymbol{\tau}}) \to \boldsymbol{\tau}' \mid \langle \boldsymbol{\tau}_1, \ldots, \boldsymbol{\tau}_n \rangle \mid \boldsymbol{\alpha} \mid \exists \boldsymbol{\alpha}.\, \boldsymbol{\tau} \\
\textit{Values} & \mathbf{v} & ::= & \mathbf{x} \mid \mathbf{true} \mid \mathbf{false} \mid \boldsymbol{\lambda}(\overline{\mathbf{x} : \boldsymbol{\tau}}).\, \mathbf{e} \mid \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle \mid \mathbf{pack}(\boldsymbol{\tau}, \mathbf{v})\ \mathbf{as}\ \exists \boldsymbol{\alpha}.\, \boldsymbol{\tau}' \\
\textit{Expressions} & \mathbf{e} & ::= & \mathbf{v} \mid \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2 \mid \mathbf{e}_1(\overline{\mathbf{e}}) \mid \langle \mathbf{e}_1, \ldots, \mathbf{e}_n \rangle \mid \boldsymbol{\pi}_i\, \mathbf{e} \mid \\
& & & \mathbf{pack}(\boldsymbol{\tau}, \mathbf{e})\ \mathbf{as}\ \exists \boldsymbol{\alpha}.\, \boldsymbol{\tau}' \mid \mathbf{unpack}(\boldsymbol{\alpha}, \mathbf{x}) = \mathbf{e}_1\ \mathbf{in}\ \mathbf{e}_2
\end{array}
$$

The typing rules are standard (with judgments $\Delta; \Gamma \vdash \mathbf{e} : \boldsymbol{\tau}$), with one exception: since language T is the target of closure conversion, we ensure via the function typing rule that functions contain no free term variables. Thus, the typing rule for functions is as follows:

$$
\frac{\cdot; \overline{\mathbf{x} : \boldsymbol{\tau}} \vdash \mathbf{e} : \boldsymbol{\tau}'}{\boldsymbol{\Delta}; \boldsymbol{\Gamma} \vdash \boldsymbol{\lambda}(\overline{\mathbf{x} : \boldsymbol{\tau}}).\, \mathbf{e} : (\overline{\boldsymbol{\tau}}) \to \boldsymbol{\tau}'}
$$

$$
\begin{array}{llll}
\textit{Types} & \varphi & ::= & \sigma \mid \tau \\
\textit{Terms} & \mathsf{e} & ::= & \ldots \mid {}^{\sigma}\mathcal{ST}\,\mathsf{e} \\
 & \mathbf{e} & ::= & \ldots \mid \mathcal{TS}\,{}^{\sigma}\,\mathbf{e} \\
 & e & ::= & \mathsf{e} \mid \mathbf{e} \\
\textit{Values} & v & ::= & \mathsf{v} \mid \mathbf{v}
\end{array}
\qquad
\begin{array}{llll}
\textit{Type Environments} & \Delta & ::= & \cdot \mid \Delta, \boldsymbol{\alpha} \\
\textit{Value Environments} & \Gamma & ::= & \cdot \mid \Gamma, \mathsf{x} : \sigma \mid \Gamma, \mathbf{x} : \tau
\end{array}
$$

$\boxed{\Delta; \Gamma \vdash e : \varphi}$

$$
\ldots \quad \frac{\Delta; \Gamma \vdash \mathbf{e} : \sigma^{+}}{\Delta; \Gamma \vdash {}^{\sigma}\mathcal{ST}\,\mathbf{e} : \sigma} \qquad \frac{\Delta; \Gamma \vdash \mathsf{e} : \sigma}{\Delta; \Gamma \vdash \mathcal{TS}\,{}^{\sigma}\,\mathsf{e} : \sigma^{+}}
$$

$\boxed{e \longmapsto e'}$

$$
\begin{array}{rcl}
{}^{\mathsf{bool}}\mathcal{ST}\,\mathbf{true} & \longmapsto & \mathsf{true} \\
{}^{\mathsf{bool}}\mathcal{ST}\,\mathbf{false} & \longmapsto & \mathsf{false} \\
{}^{\sigma_1 \to \sigma_2}\mathcal{ST}\,\mathbf{v} & \longmapsto & \lambda \mathsf{x} : \sigma_1.\ {}^{\sigma_2}\mathcal{ST}\,(\mathbf{unpack}(\boldsymbol{\alpha}, \mathbf{y}) = \mathbf{v}\ \mathbf{in}\ \boldsymbol{\pi_1}\,\mathbf{y}\ (\boldsymbol{\pi_2}\,\mathbf{y}, \mathcal{TS}\,{}^{\sigma_1}\,\mathsf{x})) \\
 & & \\
\mathcal{TS}\,{}^{\mathsf{bool}}\,\mathsf{true} & \longmapsto & \mathbf{true} \\
\mathcal{TS}\,{}^{\mathsf{bool}}\,\mathsf{false} & \longmapsto & \mathbf{false} \\
\mathcal{TS}\,{}^{\sigma_1 \to \sigma_2}\,\mathsf{v} & \longmapsto & \mathbf{pack}(\langle\rangle, \langle \mathbf{v}, \langle\rangle\rangle)\ \mathbf{as}\ \exists \boldsymbol{\alpha}.\langle\langle \boldsymbol{\alpha}, \sigma_1{}^{+}\rangle \to \sigma_2{}^{+}, \boldsymbol{\alpha}\rangle \\
 & & \quad \text{where } \mathbf{v} = \boldsymbol{\lambda}(\mathbf{z} : \langle\rangle, \mathbf{x} : \sigma_1{}^{+}).\mathcal{TS}\,{}^{\sigma_2}\,(\mathsf{v}\ {}^{\sigma_1}\mathcal{ST}\,\mathbf{x})
\end{array}
$$

Figure 2 ST: Extensions to S and T syntax, static semantics, dynamic semantics.

Closure conversion maps source terms of type σ to target terms of type σ^{+}. The definition of the type translation (σ^{+}) and environment translation (Γ^{+}) is as follows:

$$
\begin{array}{rcl}
(\mathsf{bool})^{+} & = & \mathbf{bool} \\
(\sigma_1 \to \sigma_2)^{+} & = & \exists \boldsymbol{\alpha}.\langle\langle \boldsymbol{\alpha}, \sigma_1{}^{+}\rangle \to \sigma_2{}^{+}, \boldsymbol{\alpha}\rangle
\end{array}
\qquad
\begin{array}{rcl}
(\cdot)^{+} & = & \cdot \\
(\Gamma, \mathsf{x} : \sigma)^{+} & = & \Gamma^{+}, \mathbf{x} : \sigma^{+}
\end{array}
$$

We then define a type-directed term translation $\Gamma \vdash \mathsf{e} : \sigma \rightsquigarrow \mathbf{e}$ that translates a term e such that $\Gamma \vdash \mathsf{e} : \sigma$ into a target term $\mathbf{e}$ such that $\cdot; \Gamma^{+} \vdash \mathbf{e} : \sigma^{+}$. For instance, x translates to $\mathbf{x}$ and below we show how functions are translated into closures. (We elide the rest of the translation as it is standard, e.g., see [34, 36, 45].)

$$
\frac{\begin{array}{c}\mathsf{y}_1, \ldots, \mathsf{y}_m = \text{free-vars}(\lambda \mathsf{x} : \sigma.\, \mathsf{e}) \qquad \Gamma \vdash \mathsf{y}_1 : \sigma_1 \ldots \Gamma \vdash \mathsf{y}_m : \sigma_m \qquad \boldsymbol{\tau}_{env} = \langle \sigma_1{}^{+}, \ldots, \sigma_m{}^{+}\rangle \\ \Gamma, \mathsf{x} : \sigma \vdash \mathsf{e} : \sigma' \rightsquigarrow \mathbf{e} \qquad \mathbf{v} = \boldsymbol{\lambda}(\mathbf{z} : \boldsymbol{\tau}_{env}, \mathbf{x} : \sigma^{+}).\, \mathbf{e}[\boldsymbol{\pi_1}\,\mathbf{z}/\mathbf{y}_1] \ldots [\boldsymbol{\pi_m}\,\mathbf{z}/\mathbf{y}_m]\end{array}}{\Gamma \vdash \lambda \mathsf{x} : \sigma.\, \mathsf{e} : \sigma \to \sigma' \rightsquigarrow \mathbf{pack}(\boldsymbol{\tau}_{env}, \langle \mathbf{v}, \langle \mathbf{y}_1, \ldots, \mathbf{y}_m\rangle\rangle)\ \mathbf{as}\ \exists \boldsymbol{\alpha}.\, \langle\langle \boldsymbol{\alpha}, \sigma^{+}\rangle \to \sigma'{}^{+}, \boldsymbol{\alpha}\rangle}
$$

Step 2: Define interoperability semantics. The language ST embeds the languages S and T so that both languages have natural access to foreign values (i.e., values from the other language). They receive foreign boolean values as native values, and can call foreign functions as native functions. We extend the original core languages with boundary terms ${}^{\sigma}\mathcal{ST}\,\mathbf{e}$ and $\mathcal{TS}\,{}^{\sigma}\,\mathsf{e}$ which mediate between the types σ on the source side and σ^{+} on the target side. Figure 2 presents the new syntax, typing rules and reduction rules. Typing judgments for ST have the form $\Delta; \Gamma \vdash e : \varphi$ where the environment Γ now tracks both source variables of type σ and target variables of type τ. The typing rules include all the S typing rules, but augmented with the additional environment Δ; all the T typing rules, unchanged; and rules for the two boundary constructs, shown in Figure 2.

We evaluate under a boundary until we have a value. The reduction rules for boundaries annotated bool convert boolean values from one language to the other. To convert functions across languages, we use native proxy functions. We represent a target function $\mathbf{v}$ in the source at type $\sigma_1 \to \sigma_2$ by a new function that takes an argument of type σ_1 and first

translates this argument from source to target, then unpacks the closure $\mathbf{v}$ and applies the code to its environment and the translated argument, and finally translates the result back to source at type σ_2. Notice that the direction of the conversion (and the boundary used) reverses for function arguments.

Converting source functions to target functions is a bit more subtle. To represent a source function v in the target at type $(\sigma_1 \to \sigma_2)^+$ we have to construct a closure. Since these are reduction rules, and since we only run closed programs, we know that v is closed. Hence, we simply use an empty tuple type $\langle\rangle$ for the closure environment. The underlying function $\mathbf{v}$ for this closure takes an environment of type $\langle\rangle$ and an argument of type ${\sigma_1}^+$, translates the argument to source, applies v to the translated argument, and finally translates the result back to target. The use of an empty environment in this reduction rule illustrates the difference between the translation done by boundaries in the multi-language and that done by the compiler itself.

Step 3: Define logical relation for combined language ($\approx_{ST}^{log}$) and prove $\approx_{ST}^{log} \equiv \approx_{ST}^{ctx}$. Next, we define a logical relation for the combined language ST and prove it sound and complete with respect to contextual equivalence ($\approx_{ST}^{ctx}$). In the process, we must prove the *boundary cancellation* property described in §3.1 – that is, this is the stage at which we are required to prove that the combined language can sensibly serve as an interoperability semantics.

Step 4: Prove translation is semantics preserving. If $\Gamma \vdash \mathsf{e} : \sigma \rightsquigarrow \mathbf{e}$ then $\cdot; \Gamma \vdash \mathsf{e} \approx_{ST}^{log} {}^{\sigma}\mathcal{ST}\,\mathbf{e'} : \sigma$, where $\mathbf{e'}$ is $\mathbf{e}$ with all $\mathbf{x}$ that are translations of $\mathsf{x} : \sigma' \in \Gamma$ replaced with $\mathcal{TS}^{\sigma'}\,\mathsf{x}$. Since $\approx_{ST}^{log}$ exactly captures $\approx_{ST}^{ctx}$, this lemma immediately yields the compiler correctness theorem we want.

4 Research Plan and Central Challenges

We plan to carry out this work in three stages, with the target language GTVM growing in features and functionality across the three stages. Below we discuss the main tasks and the challenges we anticipate, and describe some of the work done to date.

Verified compiler: ML to TTVM. In the first phase, we plan to develop a verified compiler from an idealized ML to a statically type-safe LLVM IR. The LLVM IR is a platform-independent, static single assignment (SSA) language [54]. An LLVM compiler normally translates a high-level language into LLVM IR. This can then be optimized using a series of IR to IR transformations. LLVM provides a collection of such transformations which perform optimizations and static analyses. The resulting LLVM IR can then be translated to a target architecture using LLVM's code generator or JIT-compiler.

LLVM programs consist of modules, which in turn contain function definitions and declarations. A function consists of a sequence of basic blocks, which as usual are a sequence of commands ending with a branch or return (`ret`) instruction. Commands include `load`, `store`, `malloc`, and `free` instructions; `alloca` for stack allocation; and a function `call` instruction. To be well formed, an LLVM program must be in valid SSA form. All components in the language are annotated with types, but the LLVM IR is not a type-safe language – like C, it allows arbitrary casts, invalid memory access, and so on.

Vellvm [57] provides a mechanized formal semantics of LLVM's IR, its static semantics, and SSA form, all formalized in Coq, as well as a proof of static safety (modulo reaching

known stuck states) via preservation and progress theorems. It says that if the program takes a step then it continues to be well formed SSA; and if the program is well formed then either it can take a step or it is in one of the defined set of stuck states. Vellvm also provides a set of tools to extract LLVM IR from Coq so it can be processed by the standard LLVM tools. Vellvm provides us with a useful starting point; without the Vellvm formalization our research plans would not be feasible.

We will first identify a statically type-safe subset of the LLVM IR (TTVM) and modify Vellvm's static safety theorems so that the progress lemma holds without the possibility of a well formed program configuration being stuck. This will require eliminating arbitrary casts, memory deallocation (`free`), and anything that leads to undefined behavior (`undef`) from the language. We will then extend the type system with polymorphism, existential types, and any other extensions needed to do type-preserving compilation from our idealized ML (language M) to TTVM (language T).

Tentatively, the compiler will consist of four languages and three passes: a closure conversion pass from M to C, an explicit allocation pass (where the data representation strategy is made explicit) from C to A, and code generation pass from A to T. To state compiler correctness, we will embed all four of the compiler's languages into a combined language $MCAT$ by defining interoperability between the adjacent languages in the compilation pipeline. The design of the interoperability semantics between M and C (for the closure conversion pass) and between C and A (for the explicit allocation pass) is already well understood and we can adapt the multi-language and logical relation from our recent work [45] which covers closure conversion and explicit allocation for System F with recursive types. Moreover, in recent work with Phillip Mates and James Perconti, we have already proved compositional correctness of closure conversion in the presence of ML-style mutable references. The presence of mutable references required a novel extension to our logical relation for the multi-language system, which we plan to report on in the near future.

We anticipate that the design of an interoperability semantics between language A and TTVM (i.e., for the code generation pass) will be the most challenging. In the language A, code still has a compositional structure even though tuples and closures are allocated on the heap – that is, a *component* is a simply a term e_A. However, at the TTVM level, that compositional structure is lost. To define interoperability between A components and T (TTVM) components, we first have to identify what exactly constitutes a TTVM *component* – that is, since there are no "terms" in TTVM, what is the shape of an e_T that we can put under a boundary $\mathcal{AT}e_T$?

Fortunately, in preliminary work with Perconti, we have already answered this question in the context of an idealized typed assembly language (TAL), which is even lower level that TTVM. For the purpose of the multi-language semantics, a TAL (or TTVM) *component* is comprised of a number of basic blocks. Thus, e_T denotes a pair $(b_0, \overline{b})$ of the currently executing basic block b_0 and the rest of the blocks $\overline{b}$ that comprise that component (which corresponds now to a TTVM function body). The next question is how do we run the term $\mathcal{AT}e_T$? As in §3, we want to run e_T until we have a value v_T and then convert that value to the language A. But running the e_T in TTVM will ultimately end with a return instruction. How do we distinguish between a normal *return within TTVM* from a *return to language A*? The solution is to introduce a special `ret-to-A` pseudo-instruction as part of the extensions we make when defining the multi-language semantics. When $\mathcal{AT}e_T$ has reduced to $\mathcal{AT}(\texttt{ret-to-A}\ v_T)$, we simply convert v_T to an A value in the usual type-directed manner. We are reasonably confident that we will be able to use ideas from our TAL work to design interoperability between A and TTVM. We do not yet know if LLVM's SSA form will complicate matters.

GTVM: a gradually type-safe LLVM IR. In the second phase, we aim to extend TTVM with support for dynamically type-safe code, assigned type `dyn`, and type-unsafe code, assigned type `un`. There is a significant body of work on contracts and gradual typing for high-level languages [19, 49, 50, 32, 6, 17, 23, 53, 44, 20, 25, 58, 24, 43, 11] that we can leverage when designing GTVM. The recent work on TS*, a gradual type system for JavaScript [52], deserves special mention as it also mixes static, dynamic, *and unsafe* types (though their dynamic type is called `any`).

As is usual in gradual type systems, interactions between type safe code and unsafe code must be protected by wrappers (dynamically checked contracts). However, note that unsafe code is just standard LLVM IR that may, for instance, be the output of a C program. To prevent raw LLVM IR (or raw C) from breaking internal invariants of statically type safe code (say from ML or Scheme), we need to ensure that C code cannot trample memory cells that belong to the type-safe parts of the language. This is one of the most significant challenges we face, but there are ideas that we can draw upon from the literature. One option is to use some sort of software-based fault isolation (as in Google's Native Client, NaCl [60]) to ensure that the unsafe code adheres to a strict *sandbox* policy, though we expect this to be too coarse-grained. Another option is to investigate whether we can devise wrappers for GTVM similar to those used by TS* (adopted from Fournet *et al.* [22]), which enforce a strict heap separation between unsafe and type-safe code. A third option is to devise contracts based on separation logic predicates similar to the approach of Agten *et al.* [1], who use these contracts to protect verified C modules from unverified C code, though this approach comes with considerable performance overhead.

As mentioned earlier, we plan to develop a compiler from GTVM to LLVM IR that inserts wrappers or safe coercions to ensure safe interoperability is preserved after the translation to LLVM IR. TS* similarly provides a compiler to JavaScript. To prove that the translation from TS* to JavaScript preserves properties such as memory isolation, the authors leverage a dependently typed version of JavaScript (called JS*) in which memory layout invariants can be specified. We conjecture that we should be able to carry out such a proof using our logical relation for GTVM which will be based on much recent progress in scaling up logical relations to so that they can be used to tractably prove sophisticated equivalences in the presence of state [2, 5, 18, 56].

Verified compiler: Rust to GTVM. In the third phase, we plan to extend the GTVM/TTVM type system so that it can support type-preserving compilation from Rust, in particular, adding features capable of expressing Rust's region and ownership discipline. For the design of the TTVM type system and logical relation, we can leverage ideas from our prior work on linear and affine type systems for memory management [10, 9, 35, 7, 21, 8]. However, instead of trying to shoehorn an appropriate notion of affine types or capabilities into TTVM, it may be better to extend the TTVM type system with dependent types in the style of HTT [38, 40]. This is a more challenging task, but a type system based on HTT might be a better choice as it can be designed independent of Rust considerations and yet should be expressive enough to serve as a target of typed compilation from Rust.

5 Conclusion

Practically every software system, from safety-critical software to web browsers, uses components written in multiple programming languages and stands to benefit from verified compositional compilers. We have proposed a proof architecture for verifying compositional

compilers based on source-target interoperability and are working on demonstrating the viability of this approach. We also propose to extend LLVM – increasingly the backend of choice for modern compilers – to support compositional compilation from type-safe source languages and principled linking with less precisely typed languages. This ambitious research program consists of numerous technical challenges and we look forward to collaborating with other groups in the community on various facets of this project.

Acknowledgements. We thank the SNAPL anonymous reviewers, Andrew Appel, Matthias Blume, Matthias Felleisen, Robby Findler, Bob Harper, Xavier Leroy, Guy McCusker, Greg Morrisett, Aaron Turon, and members of IFIP WG 2.8 (Working Group on Functional Programming) for valuable feedback on various aspects of this project. Jamie Perconti and Phillip Mates did the bulk of the work completed to date. Current project members include William Bowman, Max New, and Nick Rioux. This research is supported in part by the NSF (grants CCF-1453796, CCF-1422133, and CCF-1203008) and a Google Faculty Research Award.

References

1 Pieter Agten, Bart Jacobs, and Frank Piessens. Sound modular verification of c code executing in an unverified context. In *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*, January 2015.

2 Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*, pages 69–83, March 2006.

3 Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *International Conference on Functional Programming (ICFP), Victoria, British Columbia, Canada*, pages 157–168, September 2008.

4 Amal Ahmed and Matthias Blume. An equivalence-preserving CPS translation via multi-language semantics. In *International Conference on Functional Programming (ICFP), Tokyo, Japan*, pages 431–444, September 2011.

5 Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *ACM Symposium on Principles of Programming Languages (POPL), Savannah, Georgia*, January 2009.

6 Amal Ahmed, Robert Bruce Findler, Jeremy Siek, and Philip Wadler. Blame for all. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 201–214, January 2011.

7 Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *International Conference on Functional Programming (ICFP), Tallinn, Estonia*, pages 78–91, September 2005.

8 Amal Ahmed, Matthew Fluet, and Greg Morrisett. L3 : A linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, June 2007.

9 Amal Ahmed, Limin Jia, and David Walker. Reasoning about hierarchical storage. In *IEEE Symposium on Logic in Computer Science (LICS), Ottawa, Canada*, pages 33–44, June 2003.

10 Amal Ahmed and David Walker. The logical approach to stack typing. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, pages 74–85, January 2003.

11 João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. Polymorphic contracts. In *European Symposium on Programming (ESOP)*, March 2011.

12 Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *International Conference on Functional Programming (ICFP), Edinburgh, Scotland*, September 2009.

13 Nick Benton and Chung-Kil Hur. Realizability and compositional compiler correctness for a polymorphic language. Technical Report MSR-TR-2010-62, Microsoft Research, April 2010.

14 Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified compilation for shared-memory C. In *European Symposium on Programming (ESOP)*, April 2014.

15 Adam Chlipala. A verified compiler for an impure functional language. In *ACM Symposium on Principles of Programming Languages (POPL), Madrid, Spain*, January 2010.

16 Maulik A. Dave. Compiler verification: A bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6), 2003.

17 Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *European Symposium on Programming (ESOP)*, March 2012.

18 Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4&5):477–528, 2012.

19 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP), Pittsburgh, Pennsylvania*, pages 48–59, September 2002.

20 Cormac Flanagan. Hybrid type checking. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2006.

21 Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In *European Symposium on Programming (ESOP)*, pages 7–21, March 2006.

22 Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *ACM Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 371–384, 2013.

23 Kathryn E Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2005.

24 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *ACM Symposium on Principles of Programming Languages (POPL), Madrid, Spain*, pages 353–364, January 2010.

25 Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop (Scheme)*, pages 93–104, September 2006.

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

27 Shriram Krishnamurthi, Peter Walton Hopkins, Jay Mccarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007.

28 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML : A verified implementation of ML. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, January 2014.

29 Xavier Leroy. 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*, January 2006.

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

31 Andreas Lochbihler. Verifying a compiler for Java threads. In *European Symposium on Programming (ESOP)*, March 2010.

32 Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In *European Symposium on Programming (ESOP)*, pages 16–31, March 2008.

33 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *ACM Symposium on Principles of Programming Languages (POPL), Nice, France*, pages 3–10, January 2007.

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

35 Greg Morrisett, Amal Ahmed, and Matthew Fluet. L3 : A linear language with locations. In *Typed Lambda Calculi and Applications (TLCA), Nara, Japan*, pages 293–307, April 2005.

36 Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

37 Magnus O. Myreen. Verified just-in-time compiler on x86. In *ACM Symposium on Principles of Programming Languages (POPL), Madrid, Spain*, January 2010.

38 Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable ADTs in Hoare Type Theory. In *European Symposium on Programming (ESOP)*, pages 189–204, March 2007.

39 Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, pages 165–179, 2011.

40 Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *International Conference on Functional Programming (ICFP), Victoria, British Columbia, Canada*, pages 229–240, 2006.

41 Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. Available at: `http://www.mpi-sws.org/~dreyer/papers/pilsner/paper.pdf`, February 2015.

42 Ocsigen. `http://ocsigen.org`.

43 Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. Dependent interoperability. In *Programming Languages meets Program Verification (PLPV)*, January 2012.

44 Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP International Conference on Theoretical Computer Science*, pages 437–450, August 2004.

45 James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *European Symposium on Programming (ESOP)*, April 2014.

46 Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Not.*, 38(2):57–64, February 2003.

47 Seaside. `http://seaside.st`.

48 Jaroslav Sevcik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, 2011.

49 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop (Scheme)*, pages 81–92, September 2006.

50 Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2007.

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

52 Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin M. Bierman. Gradual typing embedded securely in JavaScript. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 425–438, 2014.

53 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2012.

54 The LLVM Development Team. The LLVM reference manual. `http://llvm.org/docs/LangRef.html`.

55 Jesse Tov. Stateful contracts for affine types. In *European Symposium on Programming (ESOP)*, March 2010.

56 Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *ACM Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 201–214, January 2013.

57 Vellvm: Verifying the llvm. `http://www.cis.upenn.edu/~stevez/vellvm/`.

58 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, pages 1–16, March 2009.

59 Peng Wang, Santiago Cuellar, and Adam Chlipala. Compiler verification meets cross-language linking via data abstraction. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 2014.

60 Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

61 Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *ACM Symposium on Principles of Programming Languages (POPL), Philadelphia, Pennsylvania*, January 2012.

62 Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Seattle, Washington*, June 2013.