Research Statement — Amal Ahmed

Despite considerable progress in compiler verification in recent years, verified compilers are still proved correct under unreasonable assumptions about what the compiled code will be linked with. These assumptions range from no linking at all—i.e., that compilation units are whole programs—to linking only with code compiled by the same compiler or from the same source language. Such assumptions contradict the reality of how we use compilers in today's world where most software systems are comprised of *components* written in *different languages* compiled by *different compilers* to a common target, as well as runtime-library routines handwritten in the target language.

The restrictions on linking adopted by state-of-the-art verified compilers stem from *limitations of the proof methods* used for establishing compiler correctness. A fundamentally different proof architecture is needed to build verified compilers for a world of multi-language software, one based on proof methods that support reasoning about *components* as well as *mixed-language programs*.

My research focuses on techniques for **provably correct and secure compilation of components** with support for **linking with target code of arbitrary provenance**. Over the last decade, I have developed new formal methods—including proof techniques for program equivalence, multi-language semantics, and type systems—and shown how to assemble them into a proof architecture for building verified correct and secure compilers for realistic, statically-typed languages (with features like mutable memory, abstract data types, and recursive types).

My long-term vision is to have verified compilers from languages as different as C, ML, Rust, and Gallina—the specification language of the Coq proof assistant—to a common low-level, *gradually typed* LLVM-like target language that enforces safe interoperability between components compiled from more precisely typed, less precisely typed, and untyped source languages. In pursuit of this vision, I have made progress on nontrivial problems in a number of areas. Below I highlight my contributions to date and explain why they are essential for correct and secure compilation in the setting of multi-language software.

Proof Methods for Program Equivalence Correct compilation of *components* that supports linking with arbitrary target code is known as *compositional* compiler correctness. It requires proof methods for establishing that the behavior of a source component is *equivalent* to the behavior of the compiled component. *Logical relations* are a well-known method for proving equivalence of program components, but for decades could not be scaled to features found in practical programming languages. My work on *step-indexed logical relations* has shown how to scale the method to realistic (typed and untyped) languages with features such as ML- and Java-style mutable references, recursive types, polymorphism, and concurrency [20, 8, 6, 5, 12, 32, 33, 57]. This work has had significant impact and step-indexed logical relations are now widely used in various contexts, e.g., to prove compiler correctness, to verify concurrent code, and to establish soundness of advanced type systems. I have also shown how to extend logical relations to support reasoning about equivalence in several mixed-language settings [43, 10, 52, 51, 17, 11].

Language Interoperability I have argued that *compositional compiler correctness is a language interoperability problem* [7]: embedding a single-language fragment in a multi-language system affects the notion of program equivalence that the compiler must use when reasoning about the correctness of optimizations. But the precise notion of equivalence can be adjusted by modifying the rules governing interoperability (linking) in the multi-language system. I developed a novel specification of compositional compiler correctness that requires that if a source component s compiles to a target component t, then t linked with some arbitrary target code t' should behave the same as s interoperating with t'. To express the latter, we give a formal semantics of interoperability between (high-level) source components and (low-level) target code. My group has used *source-target multi-language semantics* for the verification of several compiler transformations [52, 10, 51, 17]. I have also worked on *gradual typing*—which ensures safe interoperability between statically and dynamically typed code—in the presence of polymorphism [43, 18, 13].

Secure Compilation Many statically-typed programming languages provide strong information-hiding guarantees to the programmer. For instance, Java guarantees that information in a private field will remain hidden from all clients of the object and *security-typed languages* guarantee that data tagged as high-security (confidential) will remain hidden from low-security clients. For such languages, we want compilation to not only be correct, but also preserve source-level security and abstraction guarantees—i.e., target clients (attackers) should not be able to learn information from compiled components that source clients may not learn from the original source component.

Unlike existing work, which achieves secure compilation using *dynamic checks* to guard interactions between compiled components and target-level clients (attackers), my work has focused on levaraging *static checks* to ensure that compiled code is only linked with target clients that respect source-level security and abstraction guarantees. Statically enforced secure compilation avoids the significant performance overhead associated with dynamic enforcement, as long as low-level clients can be verified (e.g., via type checking).

A nontrivial challenge when building secure compilers is how to *prove* that compilation preserves source-language abstractions. The proof requires showing that any target client that a compiled component may be linked with can be *back-translated* to a behaviorally equivalent source client. Back-translation—and, hence, secure compilation—has long been considered impossible for realistic compilers since their target languages usually contain features inexpressible in the source. In recent years, I have developed back-translation techniques for increasingly challenging source and target language pairs, starting with isomorphic source and target languages with polymorphism and recursive types, to recent work with target languages containing features unavailable in the source [9, 10, 25, 51]. These results put verification of realistic secure compilers within reach.

Advanced Type Systems for State I have worked on advanced type systems for state based on *substructural* types [46, 14, 36, 15, 19, 16] and dependent types (Hoare Type Theory) [48] that support rich forms of reasoning about state—e.g., about memory aliasing, reuse, encapsulation, and invariants on portions of the heap. Such reasoning will be critical for statically-enforced secure compilation down to a low-level LLVM-like target language so we can specify which target-level clients are well behaved and okay to link with. I expect to leverage these results on substructural and dependent types when designing the type system of the LLVM-like target.

1 Compiler Verification for a Multi-Language World

The key challenge in verifying compilers for today's world of multi-language software is how to formally state a compiler correctness theorem that is compositional along two dimensions. First, the theorem must guarantee correct compilation of components while allowing compiled code to be composed (linked) with target-language components of arbitrary provenance, including those compiled from other languages (*horizontal compositionality*). Second, the theorem must support verification of multi-pass compilers by composing correctness proofs for individual passes (*vertical compositionality*).

Compiler Correctness using Multi-Language Semantics [ICFP'11, ESOP'14, ICFP'14] In joint work with my student James Perconti [52], I presented the first proof architecture for verifying multi-pass compilers in the presence of inter-language linking of compiled code. This built on joint work with Matthias Blume [10] where I developed a novel way of specifying compiler correctness based on a formal semantics of interoperability between the source and target languages of the compiler.

Informally, if a source component e_S compiles to a target component e_T then compiler correctness should require that e_S and e_T are "equivalent" or "behave the same," denoted $e_S \simeq e_T$. To formally specify $e_S \simeq e_T$, consider how the compiled component is actually used: it needs to be linked with some e'_T of arbitrary provenance, creating a whole program that can be run. Note that it doesn't make sense to link with *any* e'_T; this e'_T should at least adhere to the same calling conventions that e_T does. We use *type-preserving compilation* and target-level types to ensure that e_T can only be linked with components of certain types such that after linking the whole program is well typed. 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 the target component e'_T.

To formally specify what it means to "link a source component with a target component" we formalize a *semantics of interoperability* between the source and target languages. For a multi-pass compiler, we do this in a modular fashion. Consider a compiler that consists of two passes, from 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 $\mathsf{e_I}$ of type $\tau^{\mathcal{I}}$, where $\tau^{\mathcal{I}}$ denotes the type translation of τ. The type $\tau^{\mathcal{I}}$ provides a simple means of expressing any compiler invariants about representation and/or layout of the transformed term $\mathsf{e_I}$. The second pass analogously translates I components $\mathsf{e_I}$ 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 [44]. 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 ${}^{\tau}\mathcal{SI}(\mathsf{e_I})$ allows an I component $\mathsf{e_I}$ 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.

The multi-language system must satisfy certain properties to serve as a correct specification of when a component in one embedded language should be considered equivalent to a component in another. First, the operational semantics and type system of *SIT* must be designed so that running or typechecking a program written solely in an embedded language is the same as running or checking well-typedness in *SIT*. Second, we need a *boundary cancellation* property which says that wrapping two opposite language boundaries around a component yields the same behavior as the underlying component with no boundaries—e.g., any $\mathsf{e_S} : \tau$ must be contextually equivalent to ${}^{\tau}\mathcal{SI}(\mathcal{IS}^{\tau}\mathsf{e_S})$, and any $\mathsf{e_I} : \tau^{\mathcal{I}}$ must be equivalent to $\mathcal{IS}^{\tau}({}^{\tau}\mathcal{SI}\mathsf{e_I})$. We say that two components e_1 and e_2 are 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.

We state compiler correctness using contextual equivalence: if $\mathsf{e_S} : \tau$ compiles to $\mathsf{e_I}$, then $\mathsf{e_S} \simeq \mathsf{e_I}$, where $\mathsf{e_S} \simeq \mathsf{e_I}$ is defined as $\mathsf{e_S} \approx^{ctx}_{SIT} {}^{\tau}\mathcal{SI}(\mathsf{e_I}) : \tau$. We proceed similarly for the next pass. Since contextual equivalence is transitive, our framework achieves *vertical compositionality* immediately: it is easy to combine the two correctness proofs for the individual 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. Hence, we formalize a *logical relation* for the multi-language *SIT* that corresponds to contextual equivalence and use the logical relation to prove compiler correctness.

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: $(\mathsf{e_S}\ {}^{\tau'}\mathcal{SIT}(\mathsf{e'_T})) \approx^{ctx}_{SIT} \mathcal{SIT}(\mathsf{e_T}\ \mathsf{e'_T}) : \tau$.

I developed and refined the multi-language approach to compiler verification in the following papers:

- In my ICFP'11 paper with Matthias Blume [10], I showed how to design a source-target multi-language semantics for a type-preserving translation to continuation-passing style (CPS) from simply typed lambda calculus to a polymorphic CPS intermediate representation and used it to prove correctness of the CPS translation. (I discuss the paper's focus on secure compilation in §2.)

- In my ESOP'14 paper with James Perconti [52], I demonstrated the viability of the multi-language approach to compiler correctness by verifying the first two passes of a compiler for a realistic, statically-typed, functional language supporting higher-order functions, recursive types, polymorphism, and existential types (i.e., ADTs). The first pass performs closure conversion and the second allocates closures and data on the heap. Our target language supports imperative features (general mutable references) that are unavailable in the source language and the compiler permits linking with imperative target code that cannot be expressed in the source. The central challenge was designing a multi-language semantics (and logical relation) that preserves the information hiding and representation independence guarantees provided by type abstraction in the source and target languages.

- In an ICFP'16 paper with my students Max New and William Bowman [51], I used the multi-language approach to prove correctness of closure conversion from a functional source language without control effects to a target language with exceptions. We designed the compiler and multi-language semantics to permit linking with target components that may throw exceptions but must also handle them.

The above three papers demonstrate that our multi-language approach to compiler verification supports reasoning about (1) linking with code whose behavior could not be expressed in the compiler's source language and (2) linking with code that might be better implemented in a different language—e.g. more efficiently, or with better handling of exceptional behavior.

Supporting Mutable References and Assembly Two difficult questions left open by our ESOP'14 work were how to extend the approach to handle compilation of a source language with *mutable references* and how one would define a multi-language that embeds an *assembly language*.

In joint work with my students Phillip Mates and James Perconti [17], I have shown how to extend our ESOP'14 work to an ML-style source language with mutable references. We verified a closure conversion pass that translates from an an ML-like imperative source language without control effects to an imperative target language with first-class continuations (specifically call/cc). Our compiler correctness theorem allows compiled code (without control effects) to be linked with code that can use call/cc to manipulate control flow of the compiled component. A highly useful

example of this is linking with a library for cooperative multi-threading (an implementation of "green threads"). We designed a multi-language semantics that allows references in one language to be updated and read appropriately by code in the other language and that allows call/cc to capture a mixed-language continuation. A key contribution is the design of the multi-language logical relation which was semantically challenging due to the mix of parametric polymorphism and mutable state in both of the interoperating languages.

In joint work with Christos Dimoulas, and my students Daniel Patterson and James Perconti [11], I have developed a semantics for interoperability between a stack-based typed assembly language T and a high-level typed functional language F with recursive types. This was challenging due to the lack of compositional structure at the assembly level: the natural unit of computation one can execute and reason about in assembly is a single basic block, but a *component* e_T that we wish to place under a boundary $\mathcal{FT}$ may be comprised of multiple basic blocks. We addressed this by designing a *compositional* TAL T whose type system is augmented to use *return markers* to track where the return address for the currently executing component is stored. A jump to that return address signifies a return from the component while other jumps can be treated as intra-component. In the multi-language FT, a component e_T that we place under a boundary denotes a pair $(b_0, \overline{b})$ of the currently executing basic block b_0 and the rest of the basic blocks $\overline{b}$. To run $\mathcal{FT}\, e_T$ we start by loading the code $\overline{b}$ into memory and then running b_0. Intuitively, we would want to run e_T until we have a TAL value v_T and then convert that value to the language F. Running e_T will ultimately end with a return (jump) instruction, but how do we distinguish a normal *return within TAL* from a *return to language* F? To signal the latter, we introduce a special instruction $\texttt{ret r end}(\tau_{\texttt{F}})$ as part of the extensions we make when defining the multi-language semantics. When $\mathcal{FT}\, e_T$ has reduced to $\mathcal{FT}\,(\texttt{ret r end}(\tau_{\texttt{F}}))$, register r contains a T value of translation type τ_F^+ which we simply convert to an F value of type τ_F. A nontrivial contribution of this work is a step-indexed Kripke logical relation for the multi-language FT which we have proved sound and complete with respect to contextual equivalence. The logical relation is novel in that it can be used to prove equivalence of a high-level functional F component and an assembly-level imperative T component as well as of two TAL components comprised of different numbers of basic blocks. We are in the process of writing up the results for publication.

Comparison with Recent Approaches to Compositional Compiler Correctness Our ESOP'14 paper presented the first compiler correctness result that enjoyed both horizontal and vertical compositionality. There have been some subsequent papers on compositional compiler correctness but their proof methodologies have limitations as compared to our multi-language approach.

Stewart et al. [54] use *interaction semantics*, which provides an abstract specification of interoperability between source and target components, to prove compositional correctness of the CompCert C compiler. This work allows linking with any target component as long as it respects restrictions imposed by CompCert's memory model which is uniform across all intermediate languages of the compiler. It is not clear how to extend interaction semantics to accommodate compilers whose source and target languages use different memory models (e.g., ML and assembly).

Neis et al. [50] prove compositional correctness of a compiler for an ML-like language to an assembly-like language. They allow linking with only those target components that are related to some source component via a *parametric inter-language simulation* (PILS), a relation that specifies equivalence between source-language and target-language code. In practical terms, this means that code produced by a PILS-verified compiler can only be linked with target code produced either by the same compiler, or by a different verified compiler from the same source language to the same target language using the same PILS specification.

Kang et al. [41] prove correctness of a modified version of CompCert, dubbed SepCompCert, which explicitly restricts linking to only those components produced by the same compiler.

Future Work Figure 1 depicts my plans for future work which are described in detail in my **SNAPL'15** paper [7]. This project is funded by an NSF CAREER award. My goal is to develop GTVM, a *gradually typed* LLVM-like target language that supports safe interoperability between components that are statically type-safe (e.g., produced by type-preserving compilers from ML and Rust), 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) [1, 58, 59] by first developing TTVM (*tightly typed* LLVM), a statically type-safe version of the LLVM IR formalized in Vellvm and using it as a target for verified compilers for subsets of ML and Rust. For the ML compiler, we will build on our earlier work [52, 17, 11]. For the Rust compiler, TTVM will need type-system features capable of expressing Rust's region and ownership discipline for which I expect to leverage ideas from my prior work on linear and affine types for memory management [19, 16, 46, 14, 36, 15] and perhaps also my work on dependent types for mutable state in the style of Hoare Type Theory [48]. To support interoperability between code compiled from ML and Rust, TTVM will

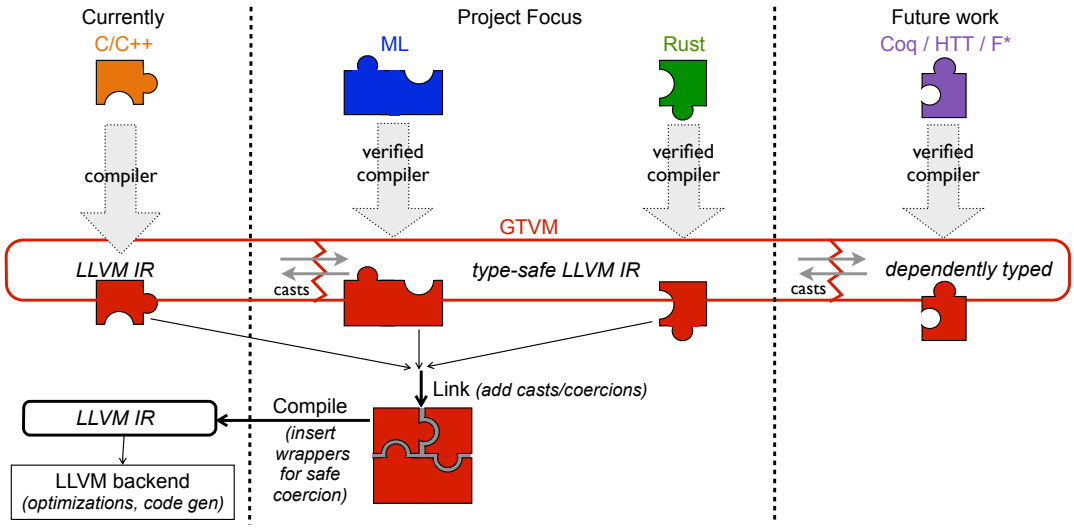

Figure 1: Research planned as part of NSF CAREER and potential future work

need static or dynamic checks to protect an affinely typed resource (from Rust) from being used more than once by a function (from ML) that knows nothing about affine types and might freely duplicate resources. With my postdoc Gabriel Scherer, I am currently investigating designs for interoperability between ML and a language with affine types to help inform this aspect of our TTVM design.

We plan to design GTVM as a gradually typed extension of TTVM. GTVM will make use of casts (i.e., 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 [56] or the verified optimizations provided by Vellvm [1, 59].

As depicted in Figure 1, my long-term vision is to extend TTVM (and GTVM) with support for dependent types to accommodate type-preserving compilers from Coq's Gallina and Microsoft's F*.

2 Secure Compilation

When building secure software systems, programmers rely on language-provided abstractions and on the assumption that any attackers—i.e., code that their software component might be linked with—will be bound by the rules of the programming language. However, after compilation, the component may be linked with target-level attackers of arbitrary provenance that may violate source-level abstractions, thus invalidating source-level security guarantees. Target attackers may be able to do things impossible in the source, such as read the compiled component's private data, modify the component's control flow, and even modify code implementing the component's methods. Secure compilation requires that we ensure that target attackers respect source-language abstractions, which requires that a compiler be *fully abstract*—i.e., it should guarantee that two source components are observationally (contextually) equivalent in the source language if and only if their compiled versions are observationally (contextually) equivalent in the target.

In much of the literature, secure (fully abstract) compilation is achieved by wrapping compiled components with dynamic checks that essentially coerce low-level attackers into behaving like high-level code. My work, however, aims to achieve secure compilation by statically ensuring that low-level program interfaces are secure. This is done by carefully devising the compiler's type translation so that any target component of translation type will have no more observational power than source-language contexts and then verifying via typechecking that we only link with target clients of translation type. This comes at the cost of disallowing linking with unverified/untyped code, but it avoids the significant performance costs of dynamically-enforced secure compilers.

Of course, statically-enforced secure compilation requires designing *statically sound* (e.g., typed) intermediate

languages for the compiler. Note, however, that fully abstract compilation is only important when compiling *components* (not whole programs) since it is a property that ensures a component is protected from an attacker context (but whole programs have no context). Hence, we only need type-preserving compilation until the intermediate language in the compiler where linking happens, yielding a whole program. At link time, we need type information in order to rule out linking with contexts that can attack the component in a way that no source context can. However, after that point in the compilation pipeline, we can erase types and do whole program compiler correctness—i.e., there is no further need for types or for proving full abstraction for the remaining passes of the compiler.

As noted earlier, proving full abstraction for realistic compilers is challenging because realistic target languages contain features unavailable in the source, while proofs of full abstraction require showing that every target context to which a compiled component may be linked can be *back-translated* to a behaviorally equivalent source context. Below I describe a series of full abstraction results that demanded increasingly powerful back-translation techniques.

Typed Closure Conversion is Fully Abstract [ICFP'08] Closure conversion is a compiler pass that collects a function's free variables in a *closure environment* that is passed as an additional argument to the function. This closed function is paired with its environment to create a *closure*. *Typed closure conversion* [45] uses an existential type to abstract the type of the environment, which ensures that two functions with the same type but different free variables still have the same type after closure conversion. In joint work with Matthias Blume [9], I showed that typed closure conversion for a polymorphic language with existential and recursive types is fully abstract. We used a target language syntactically identical to the source which made it possible to define *wrapper* terms within the language that back-translate target values of translation type τ_S^+ to source values of type τ_S. These wrappers serve as witness to a type isomorphism which allows us to prove full abstraction.

Fully Abstract CPS Translation [ICFP'11] Our ICFP'08 paper left open the question of how to do back-translation when the target language contains features unavailable in the source. In joint work with Matthias Blume [10], I presented a fully abstract CPS translation—a compiler pass that makes control flow explicit—from a simply-typed pure functional language to a polymorphic target. The CPS translation uses a locally polymorphic answer type to ensure that target components of translation type can make no more observations than source components. Since the target language is more expressive than the source (i.e., it has polymorphic types), we had to devise a new technique called *back-translation by partial evaluation*. The latter allows us to back-translate a term of translation type τ_S^+ even if the term contains sub-terms that are not of translation type (and hence not back-translatable) by performing some partial evaluation to eliminate sub-terms of non-translation type. Such partial evaluation is feasible since both source and target are terminating languages.

Translating Noninterference into Parametricity [ICFP'15] It is folklore that noninterference in security-typed languages can be encoded via parametricity but there hasn't been any work that successfully demonstrates that. In joint work with my student William Bowman [25], I presented a translation from a security-typed language—technically, the dependency core calculus (DCC) [2]—into a language with higher-order polymorphism (but no security types or lattices) and proved that the translation preserves noninterference. To express noninterference in the target, we define a notion of observer-sensitive equivalence that makes essential use of both first-order and higher-order parametric polymorphism. We show how to encode DCC's security lattice and protection judgment using a *protection ADT* encoded in the target. To prove that noninterference is preserved, we had to use a more sophisticated form of *back-translation by partial evaluation* than in our ICFP'11 work.

Fully Abstract Compilation via Universal Embedding [ICFP'16] The "back-translation by partial evaluation" technique in our ICFP'11 and ICFP'15 papers applies when the target language has more features than the source, but only if both source and target are terminating languages. In recent work with my students Max New and William Bowman [51], I presented a new proof technique for back-translation that is applicable to realistic compilers for realistic languages—that is, compilers for Turing-complete languages whose target language contains features unavailable in the source. Specifically, we proved the first full abstraction result for a translation whose target language contains exceptions but the source does not. We perform closure conversion from a functional language with recursive types to a target language that uses a modal type system to track exceptions. The type translation ensures that compiled code can only be linked with computations that are well-behaved—specifically, they may not throw unhandled exceptions. Our central contribution is a new back-translation technique, called *universal embedding* that allows this well-behaved class of target components (of translation type) to be embedded in the source language at a "universal type". Then boundaries are inserted to mediate terms between the untyped embedding and the strongly-typed source.

The technique allows back-translating non-terminating programs, target features that are untypeable in the source, and well-bracketed effects. As long as we have recursive types in both source and target, our universal embedding technique should apply when the target has other effects such as state or delimited continuations.

Future Work I plan to apply our universal embedding technique to develop a fully abstract compiler for an ML-like language to either a typed assembly language or TTVM. I also have an NSF grant to investigate secure compilation of dependently typed languages.

3 Logical Relations for Realistic Languages

Logical relations are an important proof technique for establishing many properties of programs, programming languages, and language implementations. They can be used to prove type safety, to show that one implementation of an abstract data type (ADT) can be replaced by another, and to establish the correctness of compiler transformations. Yet, for three decades, logical relations were only applied to "toy" languages because they could not be scaled to features found in practical programming languages. My dissertation [20] presented the ***first logical-relations model for a language with mutable references*** to closures or objects (i.e., "higher-order state"), a nontrivial extension of the indexed model of recursive types introduced by Appel and McAllester [21]. While ordinary logical relations are defined by induction on types and are no longer well founded in the presence of recursive types and mutable references, these *step-indexed* logical relations are defined by induction on types as well as the number of steps available for future evaluation. This stratification can handle circularities introduced by a variety of advanced typing features.

We used a model based on my thesis in the Princeton Foundational Proof-Carrying Code (FPCC) implementation to prove type safety of Typed Machine Language [8], the target of a type-preserving compiler from ML. In subsequent work with Matthew Fluet and Greg Morrisett on *substructural* type systems for state [46, 15, 14], I extended the model from my thesis to a language with both shared ML-style references as well as unique references which support explicit deallocation. These are all instances of *unary* logical relations used to prove type safety.

Logical Relations for Recursive Types and Stateful ADTs [ESOP'06, POPL'09] In my ESOP'06 paper [6], I presented the first *binary* step-indexed logical relation that can be used to reason about contextual equivalence of programs in a language with recursive types, polymorphism, and existential types (ADTs). Direct proofs of contextual equivalence are typically infeasible. Logical relations offer a tractable method for proving contextual equivalence. This method was quickly instrumental in establishing several important results [5, 9, 43] that been previously out of reach for languages with recursive types or mutable state.

Prior to 2009, though there had been some work on proving equivalence in the presence of mutable state, none of the existing methods considered languages with all of the following: polymorphism, existential types, recursive types, and higher-order state (ML/Java-style mutable references). But practical programming languages do contain all these features. In my POPL'09 paper with Derek Dreyer and Andreas Rossberg [12], I developed a novel step-indexed Kripke logical relation for proving equivalence of programs in precisely such a language. Our method can be used to prove sophisticated representation-independence results, including for stateful ADTs that define abstract types whose internal representation depends on some local state. In particular, to handle such stateful ADTs, our logical relation supports reasoning about pieces of local state that may *evolve* over time in accordance with some specified protocol.

The steps in step-indexed logical relations provide a useful induction metric, but they can also clutter proofs: to show that two programs are infinitely related, one must pick an arbitrary n and show that the programs are related for n steps. To eliminate this need for step-specific reasoning, in a **LICS'09** paper with Derek Dreyer and Lars Birkedal [32, 33], I presented a relational modal logic that essentially "hides" the steps from the user of the method, providing abstract, step-free proof principles for reasoning about relatedness in a setting with polymorphism and recursive types.

Logical Relations for Fine-Grained Concurrency [POPL'13] Coarse-grained locking of a mutable data structure provides a simple way to ensure that multiple threads can safely access it in parallel. But this essentially sequentializes all access to the data structure and thwarts any speedup one might hope to gain from parallelism. Fine-grained concurrent data structures (FCDs) reduce the granularity of critical sections in both time and space, thus making it possible for clients to access different parts of a mutable data structure in parallel. The tradeoff, however, is that implementations of FCDs—e.g., those in `java.util.concurrent`—are very subtle and tricky to reason about directly. What we want is the ability to use FCDs knowing that they behave like their coarse-grained counterparts: formally,

FCDs must be *contextual refinements* of their coarse-grained counterparts, which allows clients to reason about them as if all access to them were sequentialized.

In collaboration with Aaron Turon, Jacob Thamsborg, Lars Birkedal, and Derek Dreyer [57], I presented the first method that supports direct proofs of contextual refinement for FCDs in the setting of a type-safe, higher-order, stateful language [57]. The method extends our prior work on *logical relations* for mutable data structures [12, 34] in nontrivial ways to support refinement proofs for the most sophisticated FCDs, e.g., conditional compare-and-set (CCAS). This work formed the first half of Aaron Turon's dissertation (Ph.D., Northeastern University) which won the 2014 John C. Reynolds ACM SIGPLAN Dissertation Award.

Impact of Step-Indexed Logical Relations My work on step-indexed logical relations—in particular, my dissertation, and ESOP'06 and POPL'09 papers—has had a significant impact that goes beyond my own work. Step-indexed logical relations have been used extensively by PL researchers in a variety of contexts. Here I cite a few examples: they have been used for proofs of compiler correctness [22, 39]; for soundness of Concurrent Separation Logic [37]; for type safety of an imperative object calculus [38]; for privacy safety in a calculus for differential privacy [53]; for reasoning about modularity and data abstraction in the presence of state and control effects [34]; for type safety of a capability system for an ML-like language [23]; for normalization of the logical fragment in a language with recursion [26]; for soundness of logics for concurrent reasoning [40]; for reasoning about equivalence in a probabilistic programming language [24]; and for fully abstract compilation via approximate back-translation [31].

4 Other Work

Hoare Type Theory [ESOP'07] Modular reasoning in the presence of state is a challenging problem. With Aleks Nanevski, Greg Morrisett, and Lars Birkedal [48, 47], I presented Hoare Type Theory (HTT), a dependently-typed language that supports formally specifying and reasoning about effects, and provides a clean (monadic) separation between pure and effectful computations. HTT incorporates specifications (in the style of Hoare logic or Separation Logic) into types. The Hoare type $\{P\}x : A\{Q\}$ classifies code that can safely execute in a state satisfying the assertion P and either diverge, or terminate with a value x of type A in a state satisfying the assertion Q. The use of specifications as types has important benefits. In particular, HTT permits abstraction over specifications; this is critical for building reusable components as it allows the internal invariants of an object or module (possibly involving local state owned by the object) to be appropriately abstracted. Thus, HTT provides effective mechanisms for encapsulation in the presence of mutable state, and as a result, features such as higher-order functions, polymorphism, and abstract data types can be safely combined with heap updates and explicit memory management. This higher-order version of HTT is the foundation for Ynot, a library for the Coq proof assistant that supports verifying imperative programs [49, 30, 42]. The recent redesign of Microsoft's F* [55] is very similar to HTT except that it adds support for SMT-based automation.

Gradual Typing and Polymorphism [POPL'11] In collaboration with Philip Wadler, Robby Findler, and Jeremy Siek [13, 18], I developed a core calculus of casts between more precise and less precise types (where Dynamic is the least precise type). Our calculus supports casts to and from polymorphic types: a dynamically typed value may be cast to a polymorphic type and vice versa, with the type enforced by dynamic sealing (as in my earlier work with Matthews [43]) so as to ensure relational parametricity. Casts are akin to contracts [35] and come with a notion of blame: we have shown that when more-typed and less-typed portions of a program interact, any cast (contract) failures are the fault of the less-typed portion.

Provenance Provenance is meta-information about the origin, history, or derivation of an object. Many computer systems need to be provenance-aware in order to provide satisfactory accountability, reproducibility, and understanding of the data and processes used to compute results. Many provenance systems claim to provide such capabilities, but most lack formal definitions or guarantees of these properties. In joint work with James Cheney and Umut Acar [27, 29, 28], I presented formal provenance models for a core database query language. The first, based on the notion of *dependence* from program analysis and program slicing, formalizes how to compute what parts of the input some part of the output depends on. The second, using detailed *execution traces* as a form of provenance, shows how to *replay* traces to recompute results and how to use *trace slicing* to obtain more concise explanations of parts of the output. With Umut Acar, James Cheney, and Roly Perera [3, 4], I developed a framework for *provenance security* based on tracing semantics and formalized correct algorithms for *disclosure* of key provenance information and *obfuscation* of sensitive provenance information based on trace slicing.

References

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

[2] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 147–160, 1999.

[3] Umut Acar, **Amal Ahmed**, James Cheney, and Roly Perera. A core calculus for provenance. In *Principles of Security and Trust (POST)*, March 2012.

[4] Umut Acar, **Amal Ahmed**, James Cheney, and Roly Perera. A core calculus for provenance. *Journal of Computer Security*, 21(6):919–969, 2013.

[5] Umut A. Acar, **Amal Ahmed**, and Matthias Blume. Imperative self-adjusting computation. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 309–322, January 2008.

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

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

[8] **Amal Ahmed**, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. Semantic foundations for typed assembly languages. *ACM Transactions on Programming Languages and Systems*, 32(3):1–67, March 2010.

[9] **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.

[10] **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.

[11] **Amal Ahmed**, Christos Dimoulas, Daniel B. Patterson, and James T. Perconti. Reasonably mixing high and low: A functional language interoperating with assembly (technical report). Available at `http://www.ccs.neu.edu/home/amal/papers/funtal.pdf`, July 2016.

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

[13] **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.

[14] **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.

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

[16] **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.

[17] **Amal Ahmed**, Phillip Mates, and James T. Perconti. Under control: Compositional correctness of closure conversion with state. Available at `http://www.ccs.neu.edu/home/amal/papers/refcc.pdf`, July 2016.

[18] **Amal Ahmed**, Jacob Matthews, Robert Bruce Findler, and Philip Wadler. Blame for all. In *Workshop on Script-to-Program Evolution (STOP)*, pages 1–13, 2009.

[19] **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.

[20] Amal Jamil Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, November 2004.

[21] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.

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

[23] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Stovring, Jacob Thamsborg, and Hongseok Yang. Step-indexed kripke models over recursive worlds. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, January 2011.

[24] Ales Bizjak and Lars Birkedal. Step-indexed logical relations for probability. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 279–294, 2015.

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

[26] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Step-indexed normalization for a language with general recursion. In *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012.*, pages 25–39, 2012.

[27] James Cheney, **Amal Ahmed**, and Umut Acar. Provenance as dependency analysis. In *Proceedings of the 11th International Symposium on Database Programming Languages (DBPL), Vienna, Austria*, pages 138–152, September 2007.

[28] James Cheney, **Amal Ahmed**, and Umut Acar. Database queries that explain their work. In *ACM Conference on Principles and Practice of Declarative Programming (PPDP)*, September 2014.

[29] James Cheney, **Amal Ahmed**, and Umut A. Acar. Provenance as dependency analysis. *Mathematical Structures in Computer Science*, 21(6):1301–1337, 2011.

[30] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avi Shinnar, and Ryan Wisnesky. Effective interative proofs for higher-order imperative programs. In *International Conference on Functional Programming (ICFP)*, September 2009.

[31] Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*, 2016.

[32] Derek Dreyer, **Amal Ahmed**, and Lars Birkedal. Logical step-indexed logical relations. In *IEEE Symposium on Logic in Computer Science (LICS), Los Angeles, California*, pages 71–80, August 2009.

[33] Derek Dreyer, **Amal Ahmed**, and Lars Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2:16):1–37, June 2011.

[34] 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.

[35] 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.

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

[37] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *European Symposium on Programming (ESOP)*, Budapest, Hungary, March 2008.

[38] Cătălin Hriţcu and Jan Schwinghammer. A step-indexed semantics of imperative objects. In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, January 2008.

[39] 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.

[40] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 637–650, 2015.

[41] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*, pages 178–190. ACM, 2016.

[42] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *ACM Symposium on Principles of Programming Languages (POPL), Madrid, Spain*, pages 237–248, 2010.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] Aleksandar Nanevski, **Amal Ahmed**, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable adts in Hoare Type Theory. Technical Report TR-14-06, Harvard University, September 2006.

[48] 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.

[49] 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, 2008.

[50] 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. In *International Conference on Functional Programming (ICFP), Vancouver, British Columbia, Canada*, August 2015.

[51] Max S. New, William J. Bowman, and **Amal Ahmed**. Fully abstract compilation via universal embedding. In *International Conference on Functional Programming (ICFP), Nara, Japan*, September 2016.

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

[53] Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland*, September 2010.

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

[55] Nikhil Swamy, Ctlin Hricu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*, pages 256–270, January 2016.

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

[57] 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.

[58] 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.

[59] 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.