Research Statement

Amal Ahmed
Khoury College of Computer Sciences, Northeastern University

My research focuses on two interconnected problems: **provably correct and secure compilation** and **safe language interoperability**. Despite tremendous progress in *compiler verification*, verified compilers are still proved correct under impractical 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. When real-world linking/interoperability needs do not match the assumptions made by compiler correctness or secure compilation theorems, such linking may invalidate proofs establishing correctness of compiler transformations or proofs establishing security properties of compiled code. The situation is made worse by the fact that the foreign-function interfaces (FFIs), provided by most programming languages to support language interoperability, come with no assurance that interoperability does not violate source-language type-safety guarantees.

My research has led to new formal methods—including proof techniques for program equivalence, multi-language semantics, advanced type systems for reasoning about memory usage and low-level code, type-preserving compiler transformations that enable secure compilation, and semantics for gradual typing (which mixes static and dynamic type-checking)—and shown how to leverage them in a proof architecture for building verified correct and secure compilers for realistic, statically-typed languages (with features like mutable memory, control effects, abstract data types, and dependent types).

My long-term vision is to have verified compilers from languages as different as C, ML, Rust, Clojure, and Gallina—the specification language of the Coq proof assistant—to a common low-level, *gradually typed* WebAssembly or LLVM-like intermediate 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 recent contributions and ongoing work, and explain why it is essential for correct and secure compilation in the setting of multi-language software. (Note: "**recent work**" throughout indicates work since tenure.)

1 Compositional Compiler Correctness

Compositional compiler correctness refers to verifying correct compilation of *components*, not just *whole programs*, which means the compiler correctness theorem must support linking with (some) target code. **In recent work at ICFP'19** [53], Daniel Patterson and I identified that recent compositional compiler correctness results support a spectrum on linking options—linking with code (1) produced by the same compiler [37], (2) produced by a different compiler for the same source language [44], (3) compiled from a different language with the same expressive power [31, 63, 69], to (4) compiled from another language with greater expressive power [56]. We gave a formal CCC framework—essentially a Compositional Compiler Correctness theorem with pluggable parameters (including source and target languages, a linking set specifying what we can link with, etc.)—and showed how existing compositional compiler correctness results instantiate our CCC theorem. These instantiations shed light on the benefits and drawbacks of recent results, and provide reviewers and the compiler-verification community with a framework for understanding and comparing existing and future compositional compiler correctness theorems.

In earlier work on compositional compiler correctness for multi-language software [3, 56], we proposed a novel specification for compositional compiler correctness: 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 developed a formal semantics of interoperability between (high-level) source components and (low-level) target code as a multi-language system [43] that allows source components to be placed in target contexts and vice versa. Jamie Perconti and I showed that this *source-target multi-language semantics* approach can be used to verify multi-pass compilers by defining interoperability between the source and target of each compiler pass [56], and that the correctness proofs for passes easily compose, giving us end-to-end compiler correctness. We have used source-target multi-languages for the verification of several compiler transformations [6, 47, 56], and showed that our approach supports reasoning about linking that is outside the scope of other proof architectures, namely linking with (1) code whose behavior could not be expressed in the compiler's source language and (2) code that might be better implemented in a different language—e.g. more efficiently, or with handling of exceptional behavior.

In recent work at PPDP'19 [42], we extended our source-target multi-language approach to an ML-style imperative source language, a nontrivial challenge due to the mix of mutable references and polymorphism in both interoperating languages. We verified a typed closure conversion pass from this source, with no control effects, to an imperative target language with first-class continuations (call/cc). Our compiler correctness theorem allows compiled code (from a language without control effects) to be linked with code that can use call/cc to manipulate control flow of the compiled component. We showed a useful example of this: linking with a library for cooperative multi-threading (an implementation of "green threads").

This line of work has been recognized through invitations for **keynote addresses** (Mathematical Foundations of Programming Semantics Conference (MFPS 2019), Asian Symposium on Programming Languages (APLAS 2018), International Conference on Formal Structures of Computation and Deduction (FSCD 2016)), **invited talks** (Programming Languages Mentoring Workshop (PLMW at PLDI 2020, PLMW at ICFP 2017), and the 2019 launch of the Purdue Center for Programming Principles and Software Systems (PURPL)), **tutorial lectures** (Oregon Programming Languages Summer School (2016, 2017) and the Ph.D. School at CIRM (2014)). I have received an **NSF CAREER Award** (2015-2020) and a **Google Faculty Research Award** (2014) to support this line of work.

I have also worked on correctness of speculative optimizations, which are key to just-in-time optimization of dynamic languages. These optimizations depend on predicates about the program state, so the language implementation must monitor the validity of predicates and deoptimize the program if a predicate is invalidated. While many modern compilers rely on this approach, the interplay between optimization and deoptimization often remains opaque. **In recent work at POPL'18** [25], we designed sourir, an intermediate representation (IR) that explicitly captures the predicates and the actions required to deoptimize the program, making it easy to define correct speculative optimizations that are deoptimization aware. We show equivalence (bisimulation) of multiple versions of the same function optimized under different assumptions, proving correctness of some standard compiler optimizations, as well as three that deal with deoptimization.

Ongoing and Future Work Michael Fitzgibbons (undergraduate), Zoe Paraskevopoulou (postdoc) and I have designed RichWasm, a richly typed variant of WebAssembly (Wasm) that extends Wasm with a substructural capability-based type system for *type-safe shared memory interoperability* between Wasm modules. Notably, it also supports *safe sharing of manually managed and garbage-collected memory* between languages. We have developed type-preserving compilers to RichWasm from core ML and from L3 [8] (whose type system leverages linear capability types to safely support strong updates and manual memory management), and support safe interoperability between them. In the future, we plan to enrich RichWasm so we can perform type-preserving compilation from Rust and design a safe FFI between ML and Rust.

My postdoc Zoe Paraskevopoulou (funded by the CI Fellows 2020 program) has begun work on a platform for safe language interoperabilty based on *capability-enhanced WebAssembly built on top of CHERI*.

ARM recently announced support for the CHERI architecture which ensures safe memory usage by checking, in hardware, that each memory access is accompanied by a capability. We are designing CapWasm, a Wasm variant with runtime capability checking for memory accesses, which we will implement by compiling to CHERI to keep performance overhead low by doing checks in hardware. Thus, any Wasm program can be run with CapWasm, which will ensure memory safety. Next, we plan to design safe interoperability (sound gradual typing) between our RichWasm (with rich type system and static capability checking) and CapWasm (with dynamic, hardware-based capability checking), giving us a path to memory-safe interoperability between strongly typed languages (e.g., Rust or ML, compiled to RichWasm), and untyped or unsafe languages (e.g., C, compiled to Wasm/CapWasm).

2 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. Formally, the compiler should be *fully abstract*: it should guarantee that two source components are contextually equivalent in the source language if and only if their compiled versions are contextually equivalent in the target.

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 devising type-preserving compilers that translate source-language types appropriate target-language types, so we can leverage *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 (type-checked).

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—was long considered impossible for realistic compilers since their target languages usually contain features inexpressible in the source. In earlier work, I developed back-translation techniques for increasingly challenging source and target language pairs, where the target contains features unavailable in the source [5, 6, 13, 47]. The most recent of these results uses a technique called *universal embedding* [47], which employs recursive types in the source to support back-translation of non-terminating programs, target features that are untypeable in the source, and well-bracketed effects. This technique should scale well to verification of realistic secure compilers.

In a recent Computing Surveys (2019) paper [51] with Marco Patrignani and Dave Clarke, we present a survey of formal approaches to secure compilation from the existing literature. We focus on fully abstract compilation, the secure compilation criterion adopted by most work until quite recently, discuss static versus dynamic enforcement and the relationship with type-preserving and correct compilation. The survey fills an important need in the rapidly growing secure compilation field and already has 52 citations.

My work on secure compilation has been recognized by invitations for a **keynote address** at the Symposium on Computer Security Foundations (CSF 2020, talk titled *Secure Compilation: Challenges for the Next Decade*), and **tutorial lectures** (Oregon Programming Languages Summer School (2017, 2019)). I have also worked to build the community of researchers working on secure compilation. I co-organized the first Secure Compilation Meeting (SCM 2017) co-located with POPL—renamed the Principles of Secure

Compilation Workshop (PriSC) in 2018. I served on the PriSC PC in 2018 and 2021. I also co-organized the first Dagstuhl Seminar on Secure Compilation in 2018 (with Deepak Garg, Catalin Hritcu, Frank Piessens).

3 Type-Preserving Compilation of Dependent Types

Dependently typed languages such as Gallina—the specification language of the Coq proof assistant—are now widely used to specify and prove functional correctness of source programs. However, what we ultimately need are guarantees about correctness of compiled code. By preserving dependent types through each compiler pass, we can preserve source-level specifications and correctness proofs into the generated target-language programs. Moreover, we can use these target-level types to ensure that compiled code can never be linked with ill-behaved/insecure contexts.

Unfortunately, type-preserving compilation of dependent types is hard, in essence because dependent types allow program terms to appear in types. Dependent type systems are designed around high-level abstractions to decide the term/type equivalences needed for type checking, but compilation interferes with the type-system rules for reasoning about run-time terms. In fact, there is a negative result: in 2002, Barthe and Uustalu [11] showed that type-preserving CPS translation (continuation-passing style) is not possible for the Calculus of Constructions (CC), a subset of Gallina.

In recent work at POPL'18 [15], we showed that type-preserving CPS translation for CC is, in fact, possible. Barthe and Uustalu's negative result applies to the standard typed CPS translation, where computations of type A are assigned type $(A \to \bot) \to \bot$ after translation, which disrupts the term/type equivalence used during type-checking. Our key observation is to instead use a typed CPS translation that employs answer-type polymorphism, where CPS'd computations are assigned the type $\forall\alpha.(A \to \alpha) \to \alpha$. This type justifies, via a free theorem, a new equality rule and typing rule that we add to our CPS target language, which allow us to recover the term/type equivalences that standard typed CPS disrupts.

In recent work at PLDI'18 [14], we developed a type-preserving closure conversion translation for CC. Closure conversion translates functions with free variables into *closures*, which pair code with an environment that provides bindings for the free variables. The challenge again is to transform source-type-system rules for reasoning about a high-level abstraction (functions) to target-type-system rules for reasoning about a low-level abstraction (closures). We also prove correctness of separate compilation for both translations.

This line of work is part of my student William Bowman's PhD dissertation [12]. William is now an Assistant Professor at University of British Columbia (UBC).

Ongoing Work **In a recent submission** to POPL'22 [38], we give a type-preserving ANF translation with join-point optimization for the Extended Calculus of Constructions (ECC), including higher universes as in Gallina. ANF is a popular alternative to CPS translation in regular compilers and has benefits over CPS when compiling dependent types.

4 Language Interoperability

In recent work at PLDI'17 (FunTAL) [55] **and at FOSSACS'18** (FabULous) [59], we presented multi-language systems that mix highly disparate languages. FunTAL [55] combines a high-level typed functional language F and a stack-based typed assembly language T. 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 an assembly *component* may be comprised of multiple basic blocks. We addressed this by designing a *compositional* typed assembly language (TAL) T whose type system is augmented to track where the return address for the currently executing component is stored—i.e., in which register or at what stack index. A significant contribution of this work is a step-indexed Kripke logical relation for the multi-language FT which 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. This is the first result to guarantee safe interoperability between a high-level functional language and low-level assembly. In "FabULous" [59], we present a multi-language UL that mixes an Unrestricted language (ML) with a Linear language with linear state. UL supports in-place memory updates and safe resource handling. We prove that the embedding of ML into the multi-language is fully abstract and advocate fully abstract embedding as a design criterion for multi-languages so equivalences (and refactoring) of the embedded language remain valid even when programming in the multi-language.

In a recent paper at SNAPL'17 [52], Daniel Patterson and I proposed the idea of *linking types* to address the problem of reasoning about single-language components in a multi-lingual setting. Large software systems are often built using multiple languages with different expressive power, but a programmer writing a component of that system in one language should be able to reason about their code—e.g., about correctness of refactoring—in their language alone, without having to consider multi-language interactions. Linking types allow programmers to annotate where in a program they wish to link with components inexpressible in their unadulterated language. This enables developers to reason about (behavioral) equality using only their own language and the annotations, even though their code may be linked with code written in a language with more expressive power.

This line of work has been recognized by invitations for a **keynote address** at Strange Loop 2018 (talk titled *All the Languages Together*) and an **invited talk** at SPLASH-I 2018. I have also received an **NSF Award** (2018-2021) to investigate principled compiling and linking for multi-language software.

Ongoing and Future Work Matthews-Findler-style multi-language systems [43] have been immensely valuable for formalizing source-to-source interoperability, but they don't reflect how foreign-function interfaces (FFIs) are implemented. In practice, interoperability takes place after compilation to a common target and is mediated by "glue code" that handles *conversions* between the two languages. **In a recent submission** to POPL'22 [54], my students Daniel Patterson, Noble Mushtak (undergraduate), Andrew Wagner, and I present a novel framework for the design and verification of sound language interoperability that follows an interoperation-after-compilation strategy. We verify soundness of target-level conversions by giving a model of *source-language types* as sets of *target-language terms*. We apply the framework to verify several case studies: interoperability with (1) shared memory, (2) between a pure polymorphic language and one with strong updates, and (3) between ML and an affine language, all compiled to untyped target languages. We show how our approach helps language designers better take advantage of efficient enforcement mechanisms and opportunities for sound sharing that may not be obvious in a setting divorced from implementations.

In future work, we plan to show that our framework scales to practical FFIs, by using it to verify WebAssembly Interface Types [29] and an OCaml-Rust FFI implemented via compilation to WebAssembly.

5 Semantic Foundations for Gradual Typing

Gradually typed languages allow intermingling of statically and dynamically typed code and support gradual migration of dynamically typed code to a statically typed style. **In a series of recent papers** at ICFP'18 [46], POPL'19 [49], and POPL'20 [48], we have focused on designing gradual languages that satisfy two important criteria. The first is a stronger form of type soundness than what existing designs require: existing work focuses on just type safety, but gradual type soundness should also ensure type-based equational reasoning that justifies refactoring and compiler optimization of statically typed portions of a program. The second is support for a smooth transition from untyped to typed code, known as the *gradual guarantee* [60], or *graduality*, which says that making the types in a program more precise (changing from untyped to some type) should only add runtime checks to catch type errors, and not otherwise change program behavior.

Specifically, we present a semantic framework for the design and metatheoretic analysis of gradually typed languages based on the theory of embedding-projection pairs. We use two key ideas: (1) Design the

semantics of the gradual language by translation into a *typed language with runtime type errors*, translating the dynamic (i.e., unknown) type to a recursive tagged sum of all the possible types in the gradual language; (2) Define the semantics of casts using embedding-projection (EP) pairs, i.e., a function for embedding each values of type A into the dynamic type, and another function for projecting values from the dynamic type to A, which may error at runtime if the type is violated. EP pairs have the property that (a) embedding then projecting is the identity and (b) projecting then embedding errors more than the identity.

Although graduality has been considered an important property [60], it didn't have a clean semantic definition and was considered difficult to prove. A significant contribution of **our ICFP 2018 paper** [46] is a novel method for proving graduality using logical relations and EP pairs, which we've successfully employed in subsequent work.

Our POPL 2019 paper [49, 50] presents Gradual Type Theory (GTT), a powerful framework for the systematic design of gradual languages. With GTT, we axiomatize desired properties of gradual languages, such as graduality and type-based equational reasoning, and then derive the necessary cast (runtime) semantics. Using GTT, we can study how much freedom one has in the design of gradual languages—e.g., we have shown that if we want graduality and strong type soundness, then all casts must behave according to what is dubbed the "natural" semantics of gradual typing (which is the one used in Typed Racket).

In work at ICFP'17 [9], we presented a gradual language with polymorphism/generics that ensures parametricity (information hiding), a long-open problem. Subsequently, **our POPL 2020 paper** addresses the problem of designing a polymorphic gradual language that satisfies *both parametricity and graduality*, a significant open problem. In a POPL'19 Distinguished-Award-winning paper, Toro et al. [67] suggested *that parametricity and graduality are inherently incompatible*, at least for a traditional polymorphic typed language (System F) and opted to support only parametricity. Our POPL 2020 paper explained that the incompatibility was a result of System F syntax and showed how to devise a novel polymorphic-typing syntax and gradual language that allows us to support both parametricity and graduality. Our gradual language has a novel form of abstract types, inspired by Haskell's newtype feature and static module systems. A fundamental contribution is that we proved parametricity as a simple corollary of graduality, illustrating the connection between the two properties. Our paper also gave a counterexample showing that Toro et al.'s gradual language [67] actually violates parametricity, unless the program is fully statically typed.

This line of work is part of my student Max New's PhD dissertation [45]. Max will start as an Assistant Professor at University of Michigan, Ann Arbor this fall. This work has been recognized by an invitation for a **keynote address** (International Symposium on Principles and Practice of Declarative Programming (PPDP 2019)). I have also received an **NSF Award** (2019-2022, with Daniel Licata) to extend our GTT approach to gradual typing for effects and dependent types.

6 Ongoing and Future Work

Oxide: The Essence of Rust **In a recent submission** to POPL'22 [70] my students Aaron Weiss, Olek Gierczak, Daniel Patterson, and I present Oxide, a formal, proved sound, semantics of Rust that faithfully captures its complex borrow-checking semantics. Oxide takes a novel view of Rust lifetimes as sets of locations called *regions* which approximate the origins of references. We devise a type system for ***region-based alias management*** that tracks origin information via a control-flow-based substructural typing judgment. We hope Oxide will provide a much needed formal foundation that PL researchers can build on when developing verification tools for Rust and exploring language extensions. The only existing model for Rust is λ_{Rust} [35], which uses continuation-passing style and models the Middle IR (MIR) in the Rust compiler, and thus is too low-level to build on for projects concerned with source-level Rust. Instead of λ_{Rust}'s semantic approach to type soundness, Oxide is proved sound using progress and preservation.

Cryptis: Compositional Verification of Tagged Protocols Modern verification tools can analyze sophisticated protocols in isolation, but provide few guarantees when a protocol is composed with components that were not included in the original analysis. **In a recent submission** to POPL'22 [21], Arthur Azevedo de Amorim, Marco Gaboardi, and I present a new logic, Cryptis, for symbolic verification of protocols with *tagged* messages. Cryptis provides composition through *tag invariants* — assertions that guarantee that every message tagged in a certain way satisfies some property. Taking inspiration from separation logic, we show that if different protocols use disjoint tags, they can safely execute in parallel, even if they share private keys or other secrets. Cryptis is implemented in Coq with the Iris framework. We have used it to verify several case studies and to show that a composite system can be verified *modularly*.

Languages for Verifying and Implementing UC-Secure Protocols Cryptographers design a protocol specification and prove it secure— e.g., showing that it satisfies Universal Composability (UC) [16–18], a high standard for security since UC-secure protocols can be easily composed. Systems cryptographers then implement the protocol in an actual programming language, but with no way of formally connecting the protocol implementation and its specification. We are designing a *specification language* based on multi-party session types (MPST) [33] for writing protocols and their ideal functionalities and formally proving them UC-secure via the kinds of hybrid arguments that cryptographers typically do on paper. We are also working on an *implementation language* suited to implementing and optimizing protocols that, more importantly, integrates the specification language so one can easily ensure that the protocol implementation satisfies its UC spec. This is joint work with abhi shelat, originally funded by an **IARPA HECTOR award**, led by my student Andrew Wagner and abhi's student Jack Doerner.

Intermediate Language for Semantically Sensible Language Composition We are designing an intermediate language (IL) that is based on Call By Push Value (CBPV) [41] but comes with a low-level CompCert memory model [40]. We wish to perform type-preserving compilation from a variety of DSLs to this IL and benefit from the advanced type-based reasoning (about pure and effectful computation) that CBPV provides so we can compose code from different DSLs in a semantically sensible way. We also want to be able to lift LLVM IR into this IL and allow it to interoperate with the code compiled from DSLs. This work is funded by a **DARPA V-SPELLS award** whose goal is to lift parts of a legacy codebase into DSLs and do verified compilation, potentially to modern hardware that offers higher performance or security.

7 Impact of Earlier Work on Logical Relations

Logical relations are a well-known method for proving equivalence of program components and type soundness of languages, but for decades they could not be scaled to features found in practical programming languages. My dissertation and early-career work on *step-indexed Kripke 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 [1, 2, 4, 7, 10, 23, 24, 68]. More recently, I have extended the technique to multi-languages, correct/secure compilation, gradual typing as discussed above.

This work continues to have tremendous impact: step-indexed logical relations are now a de facto proof technique, used extensively by PL researchers in a wide variety of contexts, and regularly employed in results published at top-tier venues. Here I cite only a few recent examples: they have been used for proving soundness or security properties of advanced type systems [19, 20, 28, 30, 32, 35, 57, 58, 66]; for proofs of compiler correctness and secure compilation [22, 34, 44, 65]; for soundness of logics for concurrent reasoning [36] and verification of concurrent code [26, 39, 64]; and for soundness of program reasoning on capability machines [27, 61, 62].

I have been invited to give **tutorial lectures** on logical relations at the Oregon Programming Languages Summer School (OPLSS 2011, 2012, 2013, 2015, 2016, 2017). The lecture videos available online have played an important role in helping the PL community learn to use logical relations.

References

[1] 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.

[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**. 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.

[4] **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.

[5] **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.

[6] **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.

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

[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**, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: Parametricity, with and without types. In *International Conference on Functional Programming (ICFP), Oxford, England*, 2017.

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

[11] Gilles Barthe and Tarmo Uustalu. CPS translating inductive and coinductive types. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '02, pages 131–142, 2002.

[12] William J. Bowman. *Compiling with Dependent Types*. PhD thesis, Northeastern University, November 2018.

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

[14] William J. Bowman and **Amal Ahmed**. Typed closure conversion for the calculus of constructions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania*, 2018.

[15] William J. Bowman, Youyou Cong, Nick Rioux, and **Amal Ahmed**. Type-preserving cps translation of σ and π types is not not possible. In *ACM Symposium on Principles of Programming Languages (POPL), Los Angeles, California*, 2018.

[16] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive. Report 2000/067.

[17] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[18] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. Cryptology ePrint Archive. Report 2014/553.

[19] 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.

[20] Ezgi Cicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, January 2017.

[21] Arthur Azevedo de Amorim, **Amal Ahmed**, and Marco Gaboardi. Cryptis: Composition and separation for tagged protocols. Under review. Available at: `https://www.ccs.neu.edu/home/amal/submitted/cryptis.pdf`, July 2021.

[22] 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.

[23] 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.

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

[25] Olivier Flückiger, Gabriel Scherer, Ming ho Yee, Aviral Goel, , **Amal Ahmed**, and Jan Vitek. Correctness of speculative optimizations with dynamic deoptimization. In *ACM Symposium on Principles of Programming Languages (POPL), Los Angeles, California*, 2018.

[26] Dan Frumin, Robbert Krebbers, and Lars Birkedal. Compositional non-interference for fine-grained concurrent programs. In *IEEE Symposium on Security and Privacy*, 2021.

[27] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[28] Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. Scala step-by-step: Soundness for dot with step-indexed logical relations in iris. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.

[29] WebAssembly GitHub. Interface types proposal, 2020.

[30] Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[31] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 595–608, January 2015.

[32] Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. Machine-checked semantic session typing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 178–198, 2021.

[33] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, January 2008.

[34] 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.

[35] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. In *ACM Symposium on Principles of Programming Languages (POPL), Los Angeles, California*, 2018.

[36] 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.

[37] 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.

[38] Paulette Koronkevich, Ramon Rakow, **Amal Ahmed**, and William J. Bowman. ANF preserves dependent types up to extensional equality. Under review. Available at:
`https://www.ccs.neu.edu/home/amal/submitted/anfdep.pdf`, July 2021.

[39] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. A relational model of types-and-effects in higher-order concurrent separation logic. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, January 2017.

[40] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. The compcert memory model, version 2. Technical Report hal-00703441, Research Report: RR-7987, INRIA, 2012.

[41] Paul Blain Levy. *Call-by-Push-Value*. Ph. D. dissertation, Queen Mary, University of London, London, UK, March 2001.

[42] Phillip Mates, Jamie Perconti, and **Amal Ahmed**. Under control: Compositionally correct closure conversion with mutable state. In *ACM Conference on Principles and Practice of Declarative Programming (PPDP)*, 2019.

[43] 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.

[44] 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.

[45] Max S. New. *A Semantic Foundation for Sound Gradual Typing*. PhD thesis, Northeastern University, October 2020.

[46] Max S. New and **Amal Ahmed**. Graduality from embedding-projection pairs. In *International Conference on Functional Programming (ICFP), St. Louis, Missouri*, 2018.

[47] 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.

[48] Max S. New, Dustin Jamner, and **Amal Ahmed**. Graduality and parametricity: Together again for the first time. In *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, 2020.

[49] Max S. New, Daniel R. Licata, and **Amal Ahmed**. Gradual type theory. In *ACM Symposium on Principles of Programming Languages (POPL), Lisbon, Portugal*, 2019.

[50] Max S. New, Daniel R. Licata, and **Amal Ahmed**. Gradual type theory. *Journal of Functional Programming*, 2021. To appear.

[51] Marco Patrignani, **Amal Ahmed**, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 51(125):1–36, February 2019.

[52] Daniel Patterson and **Amal Ahmed**. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[53] Daniel Patterson and **Amal Ahmed**. The next 700 compiler correctness theorems (functional pearl). In *International Conference on Functional Programming (ICFP), Berlin, Germany*, 2019.

[54] Daniel Patterson, Noble Mushtak, Andrew Wagner, and **Amal Ahmed**. Semantic soundness for language interoperability. Under review. Available at: `https://www.ccs.neu.edu/home/amal/submitted/seminterop.pdf`, July 2021.

[55] Daniel Patterson, Jamie Perconti, Christos Dimoulas, and **Amal Ahmed**. FunTAL: Reasonably mixing a functional language with assembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Barcelona, Spain*, June 2017.

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

[57] Vineet Rajani and Deepak Garg. On the expressiveness and semantics of information flow types. *Journal of Computer Security*, 28(1):129–156, 2020.

[58] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.*, 4(POPL), December 2020.

[59] Gabriel Scherer, Max S. New, Nick Rioux, and **Amal Ahmed**. Fabulous interoperability between ML and a linear language. In *Foundations of Software Science and Computation Structures (FOSSACS)*, pages 146–162, 2018.

[60] Jeremy Siek, Micahel Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages*, SNAPL 2015, 2015.

[61] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a machine with local capabilities: Provably safe stack and return pointer management. In *European Symposium on Programming (ESOP)*, April 2018.

[62] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

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

[64] David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[65] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *International Conference on Functional Programming (ICFP), Nara, Japan*, September 2016.

[66] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runst. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[67] Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricity, revisited. In *ACM Symposium on Principles of Programming Languages (POPL), Lisbon, Portugal*, 2019.

[68] 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.

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

[70] Aaron Weiss, Olek Gierczak, Daniel Patterson, and **Amal Ahmed**. Oxide: The essence of Rust. Under review. Available at: `https://www.ccs.neu.edu/home/amal/submitted/oxide.pdf`, July 2021.