

Research Statement

Amal Ahmed

The goal of my research is to improve the security and reliability of software systems through the use of programming language technology. To that end, I am interested both in developing languages with more expressive type and proof systems and in enhancing and formally certifying the trustworthiness of languages and their implementations.

Large software systems consist of hundreds or thousands of components, and many of these may be of uncertain origin. To ensure reliable and secure operation, it is important to defend against faulty or malicious code. Statically-typed programming languages provide facilities for *information hiding*—type abstraction mechanisms like *existential types* (the basis of abstract data types or ADTs) and *parametric polymorphism*—that make large-scale software development feasible by allowing programmers to write modular and secure code. If access to some private implementation detail might enable an attack, then this detail is made inaccessible by hiding it behind an abstract interface (for instance, using an existential type). The theoretical justification for this comes from *relational parametricity*, a strong semantic property that guarantees *representation independence*—i.e., that the behavior of a client (or attacker) of an ADT cannot depend on the representation and implementation details hidden behind the abstract type.

Unfortunately, type abstraction does not always guarantee information hiding in practice. One issue is that in languages with references (mutable memory cells), it is possible to establish covert channels through which attackers can discover information about the “hidden” representation of an abstract type. This is possible because current type systems do not provide effective mechanisms for keeping references used internally by an ADT (called *local state*) separate from references that the rest of the program has access to. Thus, a pointer to a “hidden” reference can escape, leaking information about internal representation details. A related but orthogonal problem is that we lack methods for reasoning about parametricity and representation independence in the presence of both type abstraction *and* references. My research has addressed both of these problems with the goal of enhancing security and modularity in the presence of mutable references.

Another issue is that the use of type abstraction to guarantee information hiding merely ensures that private details are hidden from *source-level* attackers. Once the code is compiled, it may nonetheless be vulnerable to *target-level* attackers. To preclude such vulnerability, we need *security-preserving compilers* that ensure that any private details hidden from source-level attackers are also hidden from target-level attackers. Below I describe my research on this and other topics as well as directions for future work.

1 Safety, Security, and Correctness via Logical Relations

In the past few years, I have made significant contributions related to the method of *logical relations*. Logical relations are an important proof technique for establishing many properties of programs, programming languages, and language implementations. They have been used to show type safety, to show that one implementation of an abstract data type (ADT) can be replaced by another, to show that languages for information-flow security ensure confidentiality, and to establish the correctness of compiler transformations and optimizations. Yet, until quite recently, despite three decades of research and much agreement about their potential benefits, logical relations were only applied to “toy” languages because the method did not scale to important linguistic features, including recursive types, ML- and Java-style mutable references, and polymorphism. The key problem is that logical relations, normally defined by induction on types, are no longer well founded in the presence of recursive types or mutable references. The indexed model of recursive types introduced by Appel and McAllester [15], is a technique that permits simple and direct proofs based on operational reasoning. Here, logical relations are indexed not just by types but also by the number of steps available for future evaluation. This stratification has proved to be effective at handling circularities introduced by a variety of advanced typing features, all without the need for complicated machinery such as domain theory or category theory. Over the last few years, working with a diverse group of collaborators, I have extended step-indexed logical relations in various ways and made use of these extensions to establish

significant results in distinct contexts, from self-adjusting computation to multi-language systems to security-preserving compilation and more.

Proof Methods for Program Equivalence The notion of *observationally equivalent* behavior of programs lies at the heart of a multitude of programming-language properties and problems. Thus, methods for proving program equivalence are needed in a variety of contexts. For instance, proving program equivalence is essential for verifying the *correctness* of compiler optimizations and other program transformations. It is also crucial for reasoning about *data abstraction*—specifically, for establishing *representation independence*, the property that observable program behavior is independent of the representation and implementation details hidden behind an *abstract type*. Finally, proof methods for equivalence are used to establish various *security* properties, such as about the confidentiality or integrity of data in a program.

Program equivalence is generally defined in terms of *observational equivalence*. Two program fragments are observationally equivalent if they have the same observable behavior when placed in any valid program context. Unfortunately, direct proofs of observational equivalence are typically infeasible since the definition involves quantification over *all* possible contexts. Logical relations offer a tractable method for proving observational equivalence. (They can also be used to prove *parametricity* and to extract *free theorems* [37] from types, for instance, to justify certain compiler optimizations.) I have presented step-indexed logical relations that completely characterize observational equivalence in a language with recursive types and polymorphism [3]. This method has already been instrumental in establishing several important results [1, 31, 6]. This work was presented at the 2006 European Symposium on Programming (ESOP) [3].

The steps in step-indexed logical relations provide a useful induction metric, but they also clutter proofs done using the method: 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, Derek Dreyer, Lars Birkedal and I developed 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. This work appeared at the 2009 IEEE Symposium on Logic in Computer Science (LICS) [22].

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 support for *all* of the following without restriction: polymorphism, existential types, recursive types, and ML-style mutable references. But practical programming languages do contain all these features, albeit often in disguised form. With Derek Dreyer and Andreas Rossberg, I developed a novel step-indexed 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 is, ADTs that maintain some *local state* (i.e., references inaccessible to the rest of the program) and define abstract types whose internal representations are dependent on that local state. This work appeared at the 2009 Symposium on Principles of Programming Languages (POPL) [8].

Imperative Self-Adjusting Computation Self-adjusting computation enables writing programs that can automatically and *efficiently* respond to changes to their data (e.g., inputs). The idea behind the approach is to store all data that can change over time (or across runs) in *modifiable references* and to let computations construct *traces* that can drive *change propagation*. After changes have occurred, change propagation updates the result of the computation by re-evaluating only those expressions that depend on the changed data. Previous work on self-adjusting computation required that modifiable references be written at most once during execution—this restricted applicability of the technique to purely functional programs.

With Umut Acar and Matthias Blume, I developed techniques for imperative self-adjusting computation where modifiable references can be written multiple times [1]. To establish the *correctness* of the proposed change propagation algorithm, we proved that change propagation and from-scratch execution produce *observationally equivalent* results. A central challenge here is that imperative programs can create cyclic data structures, something that is not possible in the purely functional setting. To prove equivalence in the presence of cycles in memory, we formulated a novel, *untyped*, step-indexed logical relation for mutable state, leveraging step-indexing for well-foundedness. We also presented algorithms and data structures to realize an asymptotically efficient implementation and developed a prototype as a Standard ML library. This work was presented at the 2008 Symposium on Principles of Programming Languages (POPL) [1].

Secure Multi-Language Interoperability: Parametricity via Run-Time Sealing Though the problem of how to connect multiple programming languages into a single multi-language system has been studied extensively, the semantic properties of the resulting systems have largely been ignored. From a security perspective, when two languages interoperate, it is critical that the security or abstraction guarantees provided by each individual language (in isolation) be preserved in the multi-language setting.

In collaboration with Jacob Matthews, a graduate student at the University of Chicago, I showed that a foreign interface can employ run-time sealing to preserve the parametricity guarantees of a strongly-typed language (actually System F, but henceforth ML) even when interoperating with an untyped language (henceforth Scheme). The key is to use dynamic seals to protect all ML values that initially had an abstract type—i.e., whose representation should not be observed by other parts (e.g., Scheme parts) of the program. Thus, the parametricity guarantees of the typed language hold in the multi-language setting, though they are guaranteed via dynamic sealing by the untyped language. The proof makes use of two mutually dependent step-indexed logical relations, one (type-indexed) for ML and the other (untyped) for Scheme. Using this result we were able to give a scheme for implementing parametric higher-order contracts in an untyped setting. This work was presented at the 2008 European Symposium on Programming (ESOP) [31].

Security-Preserving Closure Conversion This work is part of a larger project aimed at building compilers that preserve all security assurances provided by the source language (see Future Work section). Programmers reasoning about the security properties of their code assume that all “attackers” will be bound by the rules of the source language in which the code is written. But once the code has been compiled, it must interact with target-level attackers. To preclude vulnerability to attacks launched at the level of the target language, the compiler must ensure that no target-level attacker can make more observations than any source-level attacker—that is, the compiler must preserve observational equivalence.

Closure conversion is a program transformation used by compilers to separate code from data. In collaboration with Matthias Blume, I showed that *typed closure conversion* [32] for a polymorphic language with existential and recursive types preserves observational equivalence. The proof relies on the step-indexed logical relation from my earlier work [3] and construction of certain *wrapper* terms that “back-translate” from target values to source values. Similar wrappers have been used in the work on *contracts* [24]. This work was presented at the 2008 International Conference on Functional Programming (ICFP) [6].

Earlier Work

Type Safety via a Model of Mutable State My thesis research [14], motivated by the requirements of a practical foundational proof-carrying code (FPCC) implementation, focused on how to prove type safety using *unary* logical relations for languages rich enough to serve as a target for type-preserving compilation of ML or Java. Such languages must support not just updatable references, but also universal types (for encoding ML polymorphism and Java inheritance) and existential types (for encoding ML function closures and Java objects). To ensure safety in the presence of aliasing, these languages permit only type-preserving updates—that is, each location may only be updated with values of its designated type. A model that naïvely tracks the designated type for each location will be inconsistent. Using step-indexing to resolve the inconsistency, Andrew Appel, Roberto Virga, and I developed the first model of type-invariant mutable references that could store values of *any* type (including functions, other references, recursive types, and even impredicative quantified types) [5, 14]. We used this model, suitably adapted to a von Neumann machine, in the Princeton FPCC implementation.

In subsequent work with Matthew Fluet and Greg Morrisett on *substructural type systems* for state, I extended the above model to prove type safety in the presence of both type-invariant (shared) references that may not be deallocated and type-varying (unique) references for which explicit deallocation is supported [33, 11, 10]. These models have helped clarify the connection between our “capability-threading” substructural type systems for state and Separation Logic.

Foundational Proof-Carrying Code Proof-carrying code (PCC) is a framework for mechanically verifying the safety of machine language programs. Under this framework, the code producer is required to provide a formal proof that the code satisfies some agreed-upon safety policy, usually type safety. To be confident of safety, the code consumer need only trust the proof checker, the runtime system, and the set of axioms

that form the safety policy. In traditional PCC, the safety policy includes a large set of unverified low-level typing rules. Subsequent *foundational* proof-carrying code (FPCC) systems have sought to minimize the size of the trusted computing base by *mechanically* verifying the soundness of these typing rules.

At Princeton, colleagues and I built an FPCC system that compiles core ML into Sparc machine code and simultaneously produces a safety proof in the form of a typed assembly language (LTAL) program. To avoid having to trust the LTAL typing rules, we built a machine-checkable proof of soundness for LTAL, encoded in higher-order logic. To do this in a modular fashion we designed Typed Machine Language (TML) which provides a rich set of constructors for types and instructions, and gave a semantics to LTAL using TML. Types in TML are predicates on machine states and values; the meaning of types is based on the operational semantics of the underlying machine. This model—based on the model for mutable references in my thesis—is used to establish the type safety of TML. A comprehensive account of this work appeared in ACM Transactions on Programming Languages and Systems (TOPLAS) [4].

Impact of Step-Indexed Logical Relations

Step-indexed logical relations have had a significant impact that goes beyond my own work. They have been used extensively by other researchers in a variety of contexts to establish results that have appeared at POPL, ICFP, and ESOP, among others. Here I'll cite a few representative examples. Benton and his collaborators, and more recently, Hur and Dreyer, have used them to build proofs of compiler correctness [16, 28] (ICFP '09 and POPL '11). Reed and Pierce have used step-indexed logical relations to prove that well-typedness in their calculus for differential privacy does, in fact, guarantee privacy safety [36] (ICFP '10). A model much like the one in my thesis [14] was used by Hritcu and Schwinghammer [27] (LMCS) when proving type safety of an imperative object calculus, and a similar model was used by Hobor et al. [26] (ESOP '08) to prove the soundness of Concurrent Separation Logic as part of their work on adapting Leroy's certified compiler (CompCert) to a concurrent setting. Dreyer and his collaborators have used step-indexed logical relations to reason about modularity and data abstraction in the presence of different state and control effects [23] (ICFP '10). Birkedal et al. recently used a step-indexed model to prove type safety of a capability system for an ML-like language [17] (POPL '11).

2 Type Systems for State

Almost all practical programming languages support dynamically allocated mutable storage but few provide sophisticated mechanisms for reasoning about memory. As a result, most offer *either* a guarantee of type safety (as in Java and ML) *or* flexibility in how memory can be manipulated (including support for explicit memory management, as in C), but not both. In general, richer forms of reasoning about state—for instance, about memory aliasing, reuse, encapsulation, and invariants on portions of the heap—are often critical for proving various security properties of programs, as well as for proving safety (for instance, in languages with deallocation). Below I describe my work on advanced type and proof systems for reasoning about state.

Hoare Type Theory Modular reasoning in the presence of state is a challenging problem that has been the focus of a great deal of research. Informally, to reason modularly about program components we need richer component interfaces that permit programmers to specify invariants about state and effects. With Aleks Nanevski, Greg Morrisett, and Lars Birkedal, I developed Hoare Type Theory (HTT), a dependently-typed language that makes it possible to formally specify and reason about effects, and provides a clean (monadic) separation between pure and effectful computations (so that reasoning about pure computations incurs no additional overhead). 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 . HTT can be viewed as a provably sound and compositional formalization of the core features of systems like ESC/Java, Spec#, JML, and Cyclone, which support extended static checking of programs.

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 version of HTT is the foundation for Ynot, a library for the Coq proof assistant that supports writing and verifying imperative programs [35, 21, 30]. This work was presented at the 2007 European Symposium on Programming (ESOP) [34].

Earlier Work

Substructural Type Systems Advanced type systems for state rely upon limiting the ordering and number of uses of data and operations to ensure that state is handled in a safe manner. For instance, (safely) deallocating a data structure requires that the data structure is never used in the future. To establish this property, a type system may ensure that the data structure is used *at most once*; after one use, the data structure may be safely deallocated, since there can be no further uses. A substructural type system provides the core mechanisms necessary to restrict the number and order of uses of data and operations. With Matthew Fluet and Greg Morrisett, I developed substructural type systems with support for strong (type-varying) updates, deallocation of references, storage of unique objects in shared references, temporarily treating shared references as unique (CQual’s `restrict`), and region-based memory management (including support for Cyclone’s dynamic regions and unique pointers). This body of work appeared at TLCA 2005 [33], in *Fundamenta Informaticae* [11], at the 2005 International Conference on Functional Programming (ICFP) [10], and at the 2006 European Symposium on Programming (ESOP) [25].

Logic-Based Typed Intermediate Languages With colleagues at Princeton, I developed logic-based type systems for reasoning about low-level memory management. With David Walker, I developed a substructural logic for reasoning about *adjacency* and *separation* of memory blocks, as well as *aliasing* of pointers [13]. We deployed this logic in a novel type system for a stack-based assembly language, using formulae of the logic to describe typing information for the heap, stack, and register file at each program point. The connectives of the logic provide a flexible yet concise mechanism for controlling allocation, deallocation, and access to both heap-allocated and stack-allocated data. In subsequent work with Limin Jia, we extended our logic to support reasoning about *hierarchical storage* [12] and used it to develop a type system for the region-based intermediate language used in the ML Kit compiler for Standard ML. This work was presented at the 2003 SIGPLAN workshop on Types in Language Design and Implementation (TLDI) [13] and at the 2003 IEEE Symposium on Logic in Computer Science (LICS) [12].

3 Ongoing and Future Work

Security-Preserving Compilation (aka Equivalence-Preserving Compilation) As explained above (see page 3), a security-preserving compiler is one that ensures that no target-level attacker can make more observations about some compiled code than any source-level attacker can make about the original code—that is, security-preserving compilers preserve observational equivalence. Kennedy [29] illustrates why we should care about such a property in practice: he shows how the compiler’s failure to preserve equivalence can be exploited by target-level attackers to compromise secure C[#] programs after they have been compiled to Microsoft’s CLR Intermediate Language.

To build security-preserving compilers, I believe that we must devise “clever” type translations so that the types of compiled terms can impose *well-behavedness* constraints on any target-level term that might interact with the result of the translation, thus ensuring that target-level attackers cannot violate source-level abstractions. As a first step, Matthias Blume and I have shown that the typed closure conversion phase of a compiler for a purely functional language is security preserving [6]. More recently, we have extended our results to the CPS translation and generalized our proof technique [7]. A number of challenges remain; here I’ll mention just two. First, if we are to use types at the assembly language level to impose well-behavedness constraints on target-level terms, then we need a typed assembly language (TAL) that allows invariants about state, pointers, and heaps to be specified within interfaces. For this, I intend to develop a TAL based on Hoare Type Theory [34] (see page 4). Second, we will need a proof method for program equivalence for this target language. Step-indexed logical relations are a promising approach since they have scaled well

to languages with a variety of powerful features, including state. But a TAL based on HTT would allow specifications to appear in types and we do not yet know how to build step-indexed models for such languages.

Equivalence-preserving translation has been investigated extensively in the past but with limited success. My work differs from earlier attempts in its focus on type-preserving rather than untyped translations, the use of step-indexed rather than denotational models (which did not scale well to advanced features, e.g., state), and because I have the benefit of recent developments like HTT, which I expect to be critical to this endeavor.

Integrated Static and Dynamic Typing Many large software systems today are written using untyped (or dynamically typed) scripting languages. Such languages allow rapid prototyping and fast adaptation to changing requirements, making them particularly well suited to the initial stages of software development. However, as these software systems grow, features of static typing are sorely needed, such as improved maintainability, code documentation, early error detection, and support for compilation to faster code (since types enable better compiler analyses). Languages that support *gradual typing* allow dynamically typed and statically typed code to coexist and interoperate, thus allowing programmers to slowly evolve parts of their code base from dynamically typed to statically typed.

Integrated static and dynamic typing is compelling even if one does not care about writing untyped code. Dependent type systems have become increasingly popular because they allow programmers to express very precise properties of programs. However, many of these precise properties cannot be checked statically. A good tradeoff, therefore, is to combine static checking of simple types with dynamic checking of precise types.

With Philip Wadler, Robby Findler, and Jeremy Siek, I have 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 [31]) so as to ensure relational parametricity. Casts are akin to contracts [24] and come with a notion of *blame*: we have shown that casts from a subtype to a supertype never fail, and that when more-typed and less-typed portions of a program interact any cast (contract) failures are the fault of the less-typed portion. This work will be presented at the 2011 Symposium on Principles of Programming Languages (POPL) [9].

Future challenges include extending the system to support full dependent types, and integration of *state and effects*. The latter is particularly critical if mainstream languages are to benefit from this technology. It is also particularly challenging since it would permit cast (contract) behavior to depend upon program state and allow assignments in casts to affect the behavior of the program. Correct blame assignment would also be trickier in such a setting.

Provenance Provenance is information recording the origin, derivation, or history of an object. It has been studied extensively in scientific databases and other settings due to its importance in helping scientists judge the validity, quality and integrity of data. Although many candidate definitions of provenance have been proposed, the mathematical or semantic foundations of provenance have received relatively little attention. This makes it difficult to compare approaches, evaluate their effectiveness, or argue about their validity.

To develop rigorous foundations for provenance, we should look to techniques from programming languages. In collaboration with James Cheney and Umut Acar, I have shown that the notion of *dependence*, familiar from program analysis and program slicing, can provide a formal foundation for understanding forms of provenance intended to show how (part of) the output *depends on* (parts of) its input. We gave a semantic characterization of such *dependency provenance* for a core database query language, showed that minimal dependency provenance is not computable and gave approximate tracking and analysis techniques. This work was presented at the 2007 Symposium on Database Programming Languages (DBPL) [19] and will appear in Mathematical Structures in Computer Science (MSCS) [20].

More recently, we have been investigating a foundational approach to provenance based on *provenance traces* [18, 2]. These are similar in some respects to the traces used in self-adjusting computation. Provenance traces can be viewed as explanations of the operational behavior of a database query not on just the current input but also on other possible (well-defined) inputs. We use this setup to give semantic characterizations of different forms of provenance and show how to extract existing forms of provenance from traces. We also present *trace slicing* techniques to compute *subtraces* pertinent to some part of the output (e.g., to provide users of scientific databases with more precise information on how some part of their data was computed).

References

- [1] U. A. Acar, **A. Ahmed**, and M. Blume. Imperative self-adjusting computation. In *ACM Symposium on Principles of Programming Languages (POPL)*, San Francisco, California, pages 309–322, Jan. 2008.
- [2] U. A. Acar, **A. Ahmed**, J. Cheney, and R. Perera. Self-explaining computation: A core calculus for provenance. Draft., Oct. 2010.
- [3] **A. Ahmed**. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*, pages 69–83, Vienna, Austria, Mar. 2006.
- [4] **A. Ahmed**, A. W. Appel, C. D. Richards, K. N. Swadi, G. Tan, and D. C. Wang. Semantic foundations for typed assembly languages. *ACM Transactions on Programming Languages and Systems*, 32(3):7.1–7.67, Mar. 2010.
- [5] **A. Ahmed**, A. W. Appel, and R. Virga. A stratified semantics of general references embeddable in higher-order logic. In *IEEE Symposium on Logic in Computer Science (LICS)*, Copenhagen, Denmark, pages 75–86, July 2002.
- [6] **A. Ahmed** and M. Blume. Typed closure conversion preserves observational equivalence. In *International Conference on Functional Programming (ICFP)*, Victoria, British Columbia, Canada, pages 157–168, Sept. 2008.
- [7] **A. Ahmed** and M. Blume. An equivalence-preserving cps translation via multi-language semantics. Draft., Oct. 2010.
- [8] **A. Ahmed**, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *ACM Symposium on Principles of Programming Languages (POPL)*, Savannah, Georgia, pages 340–353, Jan. 2009.
- [9] **A. Ahmed**, R. B. Findler, J. Siek, and P. Wadler. Blame for all. In *ACM Symposium on Principles of Programming Languages (POPL)*, Austin, Texas, Jan. 2011.
- [10] **A. Ahmed**, M. Fluet, and G. Morrisett. A step-indexed model of substructural state. In *International Conference on Functional Programming (ICFP)*, Tallinn, Estonia, pages 78–91, Sept. 2005.
- [11] **A. Ahmed**, M. Fluet, and G. Morrisett. L3 : A linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, June 2007.
- [12] **A. Ahmed**, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *IEEE Symposium on Logic in Computer Science (LICS)*, Ottawa, Canada, pages 33–44, June 2003.
- [13] **A. Ahmed** and D. Walker. The logical approach to stack typing. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, pages 74–85, Jan. 2003.
- [14] **A. J. Ahmed**. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, Nov. 2004.
- [15] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.
- [16] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *International Conference on Functional Programming (ICFP)*, Edinburgh, Scotland, Sept. 2009.
- [17] L. Birkedal, B. Reus, J. Schwinghammer, K. Stovring, J. Thamsborg, and H. Yang. Step-indexed kripke models over recursive worlds. In *ACM Symposium on Principles of Programming Languages (POPL)*, Austin, Texas, Jan. 2011.
- [18] J. Cheney, U. A. Acar, and **A. Ahmed**. Provenance traces. At <http://arxiv.org/abs/0812.0564>, July 2008.
- [19] J. Cheney, **A. Ahmed**, and U. Acar. Provenance as dependency analysis. In *Proceedings of the 11th International Symposium on Database Programming Languages (DBPL)*, Vienna, Austria, pages 138–152, Sept. 2007.
- [20] J. Cheney, **A. Ahmed**, and U. A. Acar. Provenance as dependency analysis. *Mathematical Structures in Computer Science (MSCS)*, Apr. 2011. Special Issue on Program Language Interference and Dependence, to appear.
- [21] A. Chlipala, G. Malecha, G. Morrisett, A. Shinhar, and R. Wisnesky. Effective inductive proofs for higher-order imperative programs. In *International Conference on Functional Programming (ICFP)*, Sept. 2009.
- [22] D. Dreyer, **A. Ahmed**, and L. Birkedal. Logical step-indexed logical relations. In *IEEE Symposium on Logic in Computer Science (LICS)*, Los Angeles, California, Aug. 2009.
- [23] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *International Conference on Functional Programming (ICFP)*, Baltimore, Maryland, Sept. 2010.

- [24] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, Pittsburgh, Pennsylvania, pages 48–59, Sept. 2002.
- [25] M. Fluet, G. Morrisett, and **A. Ahmed**. Linear regions are all you need. In *European Symposium on Programming (ESOP)*, pages 7–21, Vienna, Austria, Mar. 2006.
- [26] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *European Symposium on Programming (ESOP)*, Budapest, Hungary, Mar. 2008.
- [27] C. Hrițcu and J. Schwinghammer. A step-indexed semantics of imperative objects. *Logical Methods in Computer Science*, 2009.
- [28] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *ACM Symposium on Principles of Programming Languages (POPL)*, Austin, Texas, Jan. 2011.
- [29] A. Kennedy. Securing the .NET programming model. *Theoretical Computer Science*, 364(3):311–317, 2006.
- [30] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 2010.
- [31] J. Matthews and **A. Ahmed**. Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In *European Symposium on Programming (ESOP)*, pages 16–31, Budapest, Hungary, Mar. 2008.
- [32] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL)*, St. Petersburg Beach, Florida, pages 271–283, Jan. 1996.
- [33] G. Morrisett, **A. Ahmed**, and M. Fluet. L3 : A linear language with locations. In *Typed Lambda Calculi and Applications (TLCA)*, Nara, Japan, pages 293–307, Apr. 2005.
- [34] A. Nanevski, **A. Ahmed**, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare Type Theory. In *European Symposium on Programming (ESOP)*, pages 189–204, Braga, Portugal, Mar. 2007.
- [35] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *International Conference on Functional Programming (ICFP)*, Sept. 2008.
- [36] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *International Conference on Functional Programming (ICFP)*, Baltimore, Maryland, Sept. 2010.
- [37] P. Wadler. Theorems for free! In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, London, Sept. 1989.