Research Statement

Yuting Wang

Modern society relies so heavily on software systems that their correct operation has become a significant concern. There are many applications of software to which the absolute guarantee of correctness can be important; this is true for instance of software used in critical systems such as spacecrafts, airplanes and medical devices. The only way to obtain such a guarantee is to use formal verification, whose fundamental principle is to use mathematical specifications to formally articulate what we want of our software, to use logics to express such specifications and, eventually, to prove that the behavior of the software meets the specifications.

Motivated by the considerations above, I am interested in developing approaches to formally verify the correctness and security of software systems and in demonstrating their effectiveness in practice through real-world formalization tasks. In the former direction, I have worked on developing verification techniques and tools based on logics with powerful formalization capabilities that are realized in a variety of practical specification and reasoning systems such as Coq, Twelf, λProlog and Abella. For instance, for my doctoral dissertation I have significantly extended the reasoning capabilities of a framework that is especially effective in reasoning about formal systems. In the latter direction, I have applied the techniques and tools I developed to verify critical system software. For instance, I have worked on verification of compilers for both functional and imperative programming languages.

In the rest of this statement, I provide more details on the research I have conducted thus far and I then elaborate on what I would like to do in the future.

Reasoning Environment for Rule Based Specifications

Formal systems such as programming languages, proof systems and process calculi are common subjects for verification efforts. Such systems are often naturally described through syntax-directed rules, for example in the rubric of Structural Operational Semantics. The group within which I conducted my doctoral research and that I have continued to collaborate with has a focus on developing logics that are well-suited to formalizing and reasoning about these kinds of rule-based descriptions. A key thesis underlying our work is that a relational style of specification is ideally suited to many formalization tasks; this style naturally accommodates non-deterministic and non-total behavior that is common to many computational systems. A further observation is that many of the systems that we are interested in formalizing treat objects that have a binding structure. Our contention is that a predicate logic that uses a λ-calculus with restricted power as a vehicle for representing objects is ideally suited to this task. To validate our claims, we have developed and implemented the λProlog language for expressing relational specifications and the theorem-proving system Abella that is capable of treating inductive and co-inductive definitions for reasoning about specifications in λProlog. My contributions in this context have been twofold: I have participated in extending and improving the reasoning capabilities provided by Abella and I have analyzed the relationship between Abella and λProlog and ones like Twelf that are based on dependently typed λ-calculi. I provide insights into my specific achievements to-date below.

Reasoning about Higher-Order Relational Specifications. The rule-based descriptions of formal systems usually pertain to data objects embodying binding operators (e.g. functions and modules). They naturally involve the use of contexts for recording assumptions about binding variables (e.g. the typing contexts in typing rules). The need to encode contexts renders the relational specifications of formal systems "higher-order." λProlog is an effective language for encoding higher-order relational specifications. The logic underlying λProlog provides a logical representation of contexts through dynamic extensions of programs with formulas representing assumptions in the contexts. The Abella system provides the following approach to reasoning about λProlog specifications: it encodes the logic underlying λProlog as a fixed-point definition and reasons about the λProlog specifications through this encoding. Reasoning about higher-order relational specifications in their full form can be a complicated task because dynamically introduced formulas can be as complicated as those in the original program. Previously, Abella handled a subset of λProlog specifications, in particular, ones that caused only atomic formulas to be dynamically added. I have worked with a group of researchers to extend Abella to support reasoning about the full set of λProlog specifications. I did this by providing a full encoding of the logic underlying λProlog and developing an approach to finitely characterizing the dynamic extensions of λProlog programs [4]. I then demonstrated the usefulness of this extension to Abella through a series of examples. This work also led to a major new release of the Abella system (version 2.0.0) in Spring 2013.

Schematic Polymorphism The Abella system is based on a simply typed logic. As the system was originally implemented, formalizations in Abella that differ only in the way they apply to different types had to be repeated at each type. Together with Gopalan Nadathur, I have developed an approach that overcomes this limitation while preserving the logical basis of Abella [1]. The idea is to allow the logical notions such as terms and definitions in Abella to be parameterized by types, with the interpretations that they stand for the infinite collection of corresponding structures under the instantiation of the parameterizing types. For a parameterized theorem, we only consider proof structures that treat its parameterizing types as unanalyzable objects such that under any instantiation of these types they become the actual proofs of the simply typed theorem so instantiated. I have implemented these ideas in the latest version of Abella, version 2.0.6. As a result of this work, it is now possible to carry out polymorphic formalizations in Abella and also to reason about polymorphic specifications written in λProlog. Our method is quite lightweight since it piggy-backs on the existing mechanisms in Abella. It also provides an approach to introducing polymorphism into other theorem provers with reasoning principles similar to Abella (e.g., Twelf, Beluga and Hybrid) that do not yet support polymorphism.

Certified Transportation of Theorems. When proving a theorem about relational specifications, we usually fix a context first and then prove that this theorem holds relative to that context. To apply this theorem in another context, we need to show that it can be safely transported into the new context. The notion of *independence* gives us a handle to formalize the transportation of theorems. In a typed λ-calculus, the type A is independent of the type B if terms of type A in their normal form cannot contain terms of type B. In this case, properties about terms of type A are not affected by introduction or removal of terms of type B. I have developed an encoding of this notion of independence in a relational style by using the *formulas-as-types* principle (i.e. Curry-Howard correspondence) and a closed-world interpretation of contexts. Based on this encoding, I have developed an approach to formalizing the transportation of relations between contexts as theorems in Abella and to automatically deriving the proofs for them [3].

Extracting Proofs from the Twelf Algorithmic Checker. Relational specifications can be expressed in a manner closely related to λProlog by using dependent types in the Edinburgh Logical Framework (LF). The Twelf system builds in an ability to reason about such specifications. The basis for this is the designation of some arguments of a dependent type as inputs and the designation of the remaining arguments as outputs, thereby identifying types with non-deterministic functions. Twelf provides a set of tools for determining if particular relational specifications represent total functions when interpreted in this manner. Such verification of totality corresponds to implicit proofs of theorems represented by the dependent types. The checking of totality is algorithmic in natural and does not produce proof certificates that can be checked independently. Working with Gopalan Nadathur, I have developed a process for generating explicit proofs in a logic from totality checking for a subclass of LF specifications [5]. The long-term goal of this work is to certify "proofs" in systems like Twelf and Beluga that base their reasoning on algorithmic checking by extracting explicit proof objects from them into the logic underlying Abella.

Verified Compilation

An important consideration in the development of high-level programming languages is that they provide a more abstract view of computations and thereby make it feasible to verify program behavior. There is, however, a catch to this observation: programs written in such a language are verified using the model the language provides, whereas actual execution proceeds only after such programs have been translated by a compiler into a form that is executable on real hardware. To get programs verified end-to-end, it is therefore necessary to also prove the compilers that carry out such translations to be correct. This is an idea that has become quite feasible with the evolution of theorem-proving environments and it in fact represents a major thrust in modern research related to programming languages. My work has also addressed this topic. In particular, I have conducted research related to developing techniques for verified compilation of programs written in both functional and imperative programming languages, as I describe below.

Verified Compilation of Functional Programs In functional programming languages, it is possible to write programs or functions that take other functions as input and that nest other functions inside themselves. In describing the compilation of programs and in reasoning about this compilation, it therefore becomes necessary to have a clear and flexible way of treating functions as objects. One particular difficulty in doing this arises from the fact that it is necessary to provide a logically correct treatment of the relationship between the arguments of a function and their use within its body, i.e., the *binding structure* of the function. Most existing proof systems for formal verification provide only very primitive support for reasoning about binding structure. As a result, significant effort needs to be expended to reason about substitutions and other aspects of binding structure and this complicates the task of verifying compilers for functional languages.

In work leading to my doctoral degree [2, 6], I have collaboratively shown that much of this difficulty dissipates if we use a properly calibrated higher-order representation of syntax known as *higher-order abstract syntax* (HOAS), in particular, the version of HOAS supported by systems like Abella, Beluga, λProlog and Twelf. The reason why it dissipates is quite simple at an abstract level: using HOAS we can absorb the treatment of binding into the meta-theory of the framework, hence we obtain powerful tools for dealing with binding structure in implementation and there is also no need to reason about this treatment explicitly in each application of the tools. I have demonstrated the above points by showing the combined benefits of λProlog and Abella for verified compilation of functional programs in practice. I have shown how λProlog can be used to concisely implement a compiler for a representative functional language that reduces programs in it into a first-order form like C. This compiler contains the core transformations for compiling functional programs, including CPS conversion, closure conversion and code hoisting and it requires code that is an order of magnitude less than other proposed implementations of these transformations. Using Abella, I have then formally proved that these transformations preserve computational content. A key aspect of this work is that it clearly demonstrates that, in contrast to developments in systems such as Coq, such proofs are considerably simplified by the meta-level treatment of binding and substitution provided by HOAS and also by the fact that the implementations of the transformations in λProlog themselves have a logical character. The focus in these developments, which can be found at the URL `http://sparrow.cs.umn.edu/compilation`, has been on showing the virtues of the HOAS approach up to this point. However, the basic ideas extend naturally to the treatment of many additional aspects, such as polymorphism and imperative features, of real functional programming languages and I intend in the future to explicitly show this to be the case.

End-to-end Verified Compilation of C Programs The go-to language for programming system software is C. Therefore, it is no surprise that CompCert—the state-of-the-art verified compiler for C—plays key roles in various verification projects that aim to secure real-world system software. However, the original CompCert has limited support in end-to-end or modular verified compilation. Its target language is an abstract assembly language whose further compilation to executable machine code is carried out by using an external assembler and unverified. Furthermore, it does not have a way to protect private stack data—a prerequisite for realizing any kind of verified compilation of heterogeneous modules. Although solutions to these problems exist before, none of them support the full compilation chain of CompCert which includes advanced optimizations such as tail-call recognition and function inlining that are critical for generating code with good performance.

I have collaborated with my colleagues at Yale University to develop *Stack-Aware CompCert*, the first *complete* extension to CompCert that compiles to machine code and that supports modular protection of private stack regions [7]. We observed that CompCert is not able to generate machine code mainly because its stack is represented as an unbounded list of memory blocks which cannot possibly fit into the finite memory space on an actual machine. We solved this problem by augmenting the memory model of CompCert with an abstract data type to represent the concrete finite stack. By using this memory model in all the phases of compilation and keeping track of stack consumption using this "abstract stack," we are able to merge the stack memory blocks into a finite stack in CompCert's assembly language, and to further generate machine code by mapping the global data, the compiled code and the finite stack into a flat and finite memory space. We have also shown that all the optimizations performed by CompCert are correct in presence of a finite stack. In particular, we exploited the flexibility of the abstract stack to prove preservation of stack consumption by complicated transformations made to the concrete stack, such as reordering of allocations of stack frames made by tail-call optimization and merging of stack frames by function inlining. We also obtained a uniform policy to control stack access by augmenting the abstract stack with fine-grained permissions and proving the preservation of these permissions under compilation. Based on this uniform stack access policy, we developed Stack-Aware CompCertX, an extension to Stack-Aware CompCert that supports *contextual compilation*—compilation of open C modules in arbitrary contexts—whose effectiveness had been previously shown in the modular construction of verified OS kernels.

Future Research

The objective of my research in the next five to ten years is to continue to develop verification techniques in such a way that they can be applied to real-world programming systems. In current software engineering practice, large systems are often built by composing modules that are often heterogeneous, i.e., that are written in different programming languages which provide different levels of abstraction, that then interact with each other through well-defined interfaces. Thus, a key thrust in software verification must be on developing techniques that are applicable to open and heterogeneous modules and that allow correctness proofs to be composed so that they can be used to establish the desired properties of the complete systems. Moreover, the composition of modules does not stop at the user level. User-level programs need to be compiled into executable machine code, linked with system libraries and run on top of operating systems. Accordingly, verification techniques should scale to this end-to-end composition of user and system programs and ensure that the correctness guarantees obtained at the user level are preserved by it. In summary, to reason about realistic software systems, we need formal methods that are capable of *modular* verification of *heterogeneous* software systems with *end-to-end* correctness and security properties. There are many interesting research issues that need to be addressed towards realizing this broad goal. In the paragraphs below, I provide an indication of some of the directions I plan to address in my research program in the near future.

Comprehensive Treatment of Program Equivalence. An important concept in modular verification of programs is program equivalence, which has many forms in different verification problems. One notion of program equivalence is *contextual equivalence*. Contextually equivalent components should exhibit the same behavior in the same context. That is, they can replace each other without affecting the behavior of the system they reside in. It is a desired property that any implementation of a particular specification be contextual equivalent to each other. However, contextual equivalence is often not applicable to verification problems involving heterogeneous components. For example, in compiler verification we need to talk about the equivalence between higher-level source programs and low-level code which have vastly different kinds of contexts. Despite these differences, definitions of program equivalence often have a common set of desired properties, such as reflexivity, transitivity and composibility. For my thesis work, I have studied contextual equivalence and several definitions of program equivalence for compiler verification. In the future, I would like to do a comprehensive study of different definitions of program equivalence, and to develop techniques and tools for facilitating their formalization and application in various formal verification tasks.

Verified Compilation to Formalized Machine Architectures I believe that the abstract stack based approach to generating machine code used in Stack-Aware CompCert is quite general. For now, I have only applied it to generate 32-bits x86 machine code based on the formal model of x86-32 used in the RockSalt project that aimed to provide software fault isolation for the x86. There have been a lot of other work on formalizing computer architectures. Some notable examples are the machine models for ARM-v8, MIPS and RISC-V developed using the SAIL language at Cambridge University and the RISC-V semantics developed at MIT. I plan to connect Stack-Aware CompCert with these machine models. This work will validate the generality of our approach and, more importantly, enable linking of the compiled machine code with verified OS kernels which I will describe below.

Linking User-Level Programs with Verified OS Kernels To get end-to-end correctness guarantees of user-level programs, we need not only to show that their correctness properties are preserved after they are compiled to machine code (as Stack-Aware CompCert does) but also to show that the compiled machine code possesses the same properties when it is loaded into user processes and run on top of OS kernels. Working with my colleagues at Yale, I am exploring the linking of verified machine code with CertiKOS—a verified concurrent OS kernel developed at Yale. CertiKOS exposes a set of system calls with precise operational semantics for their low-level implementations running on actual hardware. By defining a machine language that can make system calls to CertiKOS, I aim to prove correctness and security properties of programs written in this language based on the semantics of the system calls. I will then prove that the behavior of such machine code is preserved when they are linked with the actual implementation of CertiKOS through its refinement-based correctness proof. Based on these investigations, I plan to define a flexible proof-carrying format for verified machine code (like the ELF format for binary code in Unix and Unix-like systems albeit extended with proofs) such that verified compilers such as Stack-Aware CompCert can transport user-level properties of programs into this format by verified compilation, and the compiled code in this format can be linked with verified OS kernels to eventually form end-to-end correctness guarantees.

Higher-Order Representation of Syntax with General Theorem Provers I am also interested in developing techniques to connect verification of programs written in higher-order programming languages with that of lower-level programs. The former involves manipulation and reasoning about binding structure which are important and complicated in nature and, as we have shown, can be significantly simplified by adopting a higher-order representation of syntax such as HOAS. The latter is often carried out by using general theorem provers such as Coq. However, the requirement of a weak λ-calculus in HOAS conflicts with the rich languages that general theorem provers use (e.g. Coq is based on a rich dependently-typed λ-calculi). There has been previous work on building libraries that support HOAS for general theorem provers, such as the Hybrid system. However, these libraries generally do not have the full power of HOAS in reasoning due to the conflicts previously mentioned. Using my previous experience with various systems that support HOAS, I would like to investigate the possibility of deriving the full benefits of HOAS in general theorem provers without changing their basis for reasoning. This work will enable translation of Abella proofs into proofs in general theorem provers, which in turn has interesting applications such as end-to-end verification of functional programs via connecting our verified compiler for functional languages with Stack-Aware CompCert.

Extensible Verified Proof Checkers Proof assistants like Coq provide high assurance of the properties proved in them because they generate proof terms that can be independently validated by a simple proof checker. A proof checker usually builds in a strategy which employs certain fixed decision procedures to reduce the sizes of proof terms. For instance, Coq does not generate proof terms for the equality between terms that can be decided by comparing their normal forms. The current model of proof checking has two shortcomings. First, the proof checking process itself is not verified. Second, the proof terms can be very large even with the built-in optimizations. Even though there has been previous work on building verified proof checkers, they deal with them as monolithic systems. Working with my colleagues at Yale, I plan to develop an approach to building modular, extensible and verified proof checkers. A proof checker so built will consist of modules organized as abstraction layers that can be freely composed horizontally or vertically. New layers containing decision procedures based on domain-specific knowledge can be constructed by using decision procedures in lower-level layers and will be verified through refinement-based proofs. By exploiting proof erasures techniques, we do not need to generate proof terms for these decision procedures. This will enable domain-specific optimizations of proof terms which will significantly reduce the cost of communication of proof terms and computation in proof checking.

References

[1] NADATHUR, G., AND WANG, Y. Schematic polymorphism in the abella proof assistant. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming* (New York, NY, USA, 2018), PPDP '18, ACM, pp. 15:1–15:13.

[2] WANG, Y. *A Higher-Order Abstract Syntax Approach to the Verified Compilation of Functional Programs*. PhD thesis, University of Minnesota, Dec. 2016.

[3] WANG, Y., AND CHAUDHURI, K. A proof-theoretic characterization of independence in type theory. In *Proceedings of the 13th International Conference on Typed Lambda Calculi and Applications (TLCA)* (Warsaw, Poland, July 2015), T. Altenkirch, Ed., Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 332–346.

[4] WANG, Y., CHAUDHURI, K., GACEK, A., AND NADATHUR, G. Reasoning about higher-order relational specifications. In *Proceedings of the 15th International Symposium on Princples and Practice of Declarative Programming (PPDP)* (Madrid, Spain, Sept. 2013), T. Schrijvers, Ed., pp. 157–168.

[5] WANG, Y., AND NADATHUR, G. Towards extracting explicit proofs from totality checking in twelf. In *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks and Meta-languages: Theory and Practice* (New York, NY, USA, 2013), LFMTP '13, ACM, pp. 55–66.

[6] WANG, Y., AND NADATHUR, G. A higher-order abstract syntax approach to verified transformations on functional programs. In *Programming Languages and Systems. ESOP 2016* (2016), P. Thiemann, Ed., no. 9632 in LNCS, Springer.

[7] WANG, Y., WILKE, P., AND SHAO, Z. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang. 3*, POPL (Jan. 2019), 62:1–62:30.