Research Statement

Andrei Ştefănescu

December 15, 2015

My main research interests are in Programming Languages and Formal Methods, with focus on improving software quality via program verification. Poor software quality can lead to financial loses and loss of life as demonstrated by numerous recent incidents. To achieve the high level of quality desired for critical software components, we need to formally verify these components. In particular, we need techniques that allow for easy construction of verification tools, and automatic reasoning about program correctness properties.

My results include techniques for automatically building efficient correct-by-construction program verifiers from operational semantics [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. I implemented these techniques in the $\mathbb{K}$ verification infrastructure, which in turn yielded automatic program verifiers for C, Java, and JavaScript. The Runtime Verification startup (`http://runtimeverification.com`) is currently further developing the infrastructure, and was awarded a NASA SBIR grant to apply it in the flight critical systems area. I also developed techniques for automated reasoning with a focus on proving data-structure properties [12, 13]. This resulted in the first automatic proofs of full functional correctness for a wide variety of data-structures (red-black trees, AVL trees, binomial heaps, B-trees, etc).

Operational Semantics Based Program Verification

Building program verification tools for real-world languages is hard. An operational semantics serves as the trusted model of a programming language, acting both as documentation and reference implementation. Unfortunately, it is not suitable for program verification because its operational nature leads to low-level reasoning and requires explicit induction.

Considering this limitation, there are two traditional approaches to build program verifiers for real-world languages. One approach is to give an axiomatic semantics (i.e., Hoare logic) to the target programming language (intuitively a set of proof rules). This process is both effort-intensive and error-prone (typically, an axiomatic semantics consists of more then 10,000 lines). To alleviate correctness concerns, it is common to prove such language-specific proof systems sound with respect to an operational semantics (considered a trusted model of the language), but that needs to be done for each language separately and requires extra effort. Alternatively, many program verification tools forgo defining any semantics altogether, and instead they just implement an ad-hoc verification condition generation (strongest-postcondition/weakest-precondition). This approach requires less effort but it is more error-prone, as it is complex to implement and difficult to test via execution.

Recently, there has been a lot of progress on verification decision procedures, especially in the contexts of heaps, often based on automated theorem provers. However, applying these techniques to real-world languages, like C, Java, JavaScript, etc is not straightforward.

I propose to go back to the ideal approach of leveraging existing operation semantics for program verification. We build correct-by-construction program verifiers directly from operational semantics, without defining any other semantics or verification condition generator or translator.

My insight is that many of the tricky language-specific details (like type systems, scoping, implicit conversions, etc) are orthogonal to features that make program verification hard (reasoning about heap-allocated mutable data structures, integers/bit-vectors/floating-points, etc). As such, I propose a methodology to separate the two: (1) define an operational semantics, and (2) implement reasoning in the language-independent

infrastructure. My approach has two advantages over the traditional approaches: (1) it provides a way to obtain semantics-based verifiers without a need for multiple semantics, equivalence proofs, or translators; and (2) it separates reasoning from language-specific operational details.

On the theoretical side, I proposed reachability logic [3, 4, 5, 6, 7] as a foundation achieving language-independent program verification. Specifically, I introduced one-path reachability rules $\varphi \Rightarrow^{\exists} \varphi'$, which generalize operational semantics reduction rules, and all-path reachability rules $\varphi \Rightarrow^{\forall} \varphi'$, which generalize Hoare triples. A reachability rule is a pair of formulae capturing the partial correctness intuition: for every pair (code, state) γ satisfying φ, one path ($\exists$), respectively each path ($\forall$) derived using the operational semantics from γ either diverges or otherwise reaches a pair γ' satisfying φ'. Then, I gave a language-independent proof system that derives new reachability rules (program properties) from a set of given reachability rules (the language operational semantics), at the same proof granularity and compositionality as a language-specific axiomatic semantics. The proof system consists of only 8 proof rules. I proved that the proof system is sound and relatively complete. In effect, the proof system subsumes all the language specific proof rules for loop invariants, recursive functions, etc from Hoare logic.

On the practical side, I lead the implementation effort of the $\mathbb{K}$ verification infrastructure [1, 2, 3, 5, 8, 9] based on the language-independent proof system I proposed. The framework takes an operational semantics as a parameter and uses it to automatically derive program correctness properties. In other words, the verification infrastructure *automatically* generates a program verifier from the semantics, which is *correct-by-construction* with respect to the semantics. Internally, the verifier uses the operational semantics to perform symbolic execution. Also, it has an internal prover for reasoning about program states, which makes calls to external theorem provers. A major difficulty in a language-independent setting is that standard language features relevant to verification, like control flow or memory access, are not explicit, but rather implicit (defined through the semantics). Thus, I adapted existing techniques to a language-independent setting. In particular, I noticed that symbolic execution is captured by narrowing (instead of rewriting which captures concrete execution). For reasoning about heap-manipulating data-structures, I adapted my work on natural proofs [12, 13]. The generated program verifiers are fully automated. The user only provides the program correctness specifications. The verification infrastructure is implemented in Java; it consists of approximately 30,000 non-blank lines of code, and it took about 3 man-years to complete. Several graduate and undergraduate students contributed on this project.

To ascertain the practicality of my approach, I instantiated the $\mathbb{K}$ verification infrastructure with the operational semantics of C, Java, and JavaScript (all developed independently from this project), thus obtaining program verifiers for these complex real-world languages. I evaluated these verifiers by checking the full functional correctness of challenging heap manipulation programs implementing the same data-structures in these languages (e.g. AVL trees). These programs have been used before to evaluate verification approaches. The verification time is competitive with other state-of-the-art verifiers. The time is dominated by symbolic execution, which reflects the complexity of the operational semantics and the languages themselves. Reasoning about the mathematical properties of the data-structures is similar in all three languages. Regarding the number of user annotations, my approach is comparable to the state-of-the-art language-specific approaches that do not infer invariants. Thus, my approach is effective both in terms of verification capabilities and user effort. My works has resulted in the first time that verifiers for C, Java, and JavaScript are sharing the same core infrastructure.

Natural Proofs For Program Verification

Entirely decidable logics are too restrictive to support the verification of the complex specifications of heap manipulating programs implementing data-structures. On the other hand, logics requiring manual/semi-automatic reasoning put too much burden on the user (in the form of proof tactics and lemmas).

To address these limitations, I developed the natural proofs approach [12, 13], which combines the two methodologies above. It (1) identifies a class of simple proofs for verifying heap manipulating programs, and (2) builds terminating procedures that efficiently and thoroughly search this class of proofs. This results in

a sound, incomplete, but terminating procedure that finds natural proofs automatically and efficiently.

Specifically, the program specifications are expressed using recursively defined predicates/functions. During the symbolic execution of the code, the recursive definitions are unfolded precisely across the memory footprint (the memory locations accessed by the code). Thus, the verification reduces to checking the satisfiability of a quantifier-free formula depending only on the values of predicates/functions on the frontier of the footprint. The recursive definitions are abstracted as uninterpreted functions and the resulted formula is sent to an automatic logic solver.

I evaluated my approach by verifying the full functional correctness of data-structures ranging from sorted linked lists, binary search trees, max-heaps, treaps, AVL trees, red-black trees, B-trees, and binomial heaps. This benchmarks are an almost exhaustive list of algorithms on tree-based data-structures covered in a first undergraduate course on data-structures.

Future Directions

I plan to continue developing techniques that make it easier to build sound program verifiers for real-world languages, and to automatically reason about programs in the respective languages. At the same time, I plan to apply my techniques on new verification challenges.

Improve the $\mathbb{K}$ verification infrastructure My short-term goal is to further improve the applicability of my language-independent program verification approach and of the $\mathbb{K}$ verification infrastructure. Two of the main limitations of program verification in practice are the lack of proof automation and the high user annotation burden. Thus, to address the issue of automation, I plan to extend the natural proofs techniques [12, 13] to handle increasingly complex data-structures, which are used in modern day software. Specifically, this work would address data-structures with sharing (e.g. graphs) by identifying common patterns naturally occurring in programs and how to reason about them. Also, in order to reduce the user annotation burden, I plan to develop invariant inference techniques for heap-manipulating data-structures that my verification infrastructure can already handle. Here I would build on existing techniques for more restrictive state properties, and I would adapt the techniques to the language-independent setting.

Extend my approach to other verification areas The $\mathbb{K}$ verification infrastructure allows for symbolic execution based on the operational semantics of a language. I would like to apply its symbolic execution component to other areas, such as symbolic execution based test-case generation or symbolic model checking. These areas have many techniques and tools, but suffer from fragmentation in the sense that some techniques are implemented for C (e.g. KLEE), some for Java (e.g. JavaPathFinder), some for C# (e.g. Pex), etc. Moreover, these techniques are usually not dependent on the specific languages they are implemented for, but are rather abstract in term of states and transition systems. By using my language-independent verification framework, I can make all the techniques available for all the real-world languages that have semantics, as long as the techniques are adapted to work on states specified in matching logic, and on transition specified in reachability logic. This has two advantages: (1) it would enable the researches in these areas to focus on improving the techniques, without having to worry about the language specifics, and (2) it would allow an easy way to create such tools for languages that have not been the focus of these areas (e.g. Python, JavaScript).

Apply the semantics-based verifiers to large software components My approach provides an easy way to yield program verifiers for multiple languages and platforms from a reusable model (the operational semantics). This makes it easy to handle heterogeneous systems that use several languages, and that require models for certain components. I would like to use my approach to verify critical parts of such a complex system, particularly key operating system components. By using operational semantics, it would be scalable to build the verification infrastructure for all the languages used to write the software operating all the different components, and to model the interaction between them. As part of this effort, I would build upon the state-of-the-art in terms of automation in order to make the verification effort tractable.

References

[1] Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. Under submission.

[2] Daejun Park, Andrei Stefanescu, and Grigore Rosu. KJS: A complete formal semantics of JavaScript. In *PLDI*, pages 346–356. ACM, 2015.

[3] Andrei Ştefănescu, Stefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *RTA*, volume 8560 of *LNCS*, pages 425–440, July 2014.

[4] Grigore Roşu, Andrei Ştefănescu, Stefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *LICS*, pages 358–367. IEEE, 2013.

[5] Grigore Roşu and Andrei Ştefănescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.

[6] Grigore Roşu and Andrei Ştefănescu. From Hoare logic to matching logic reachability. In *FM*, volume 7436 of *LNCS*, pages 387–402, 2012.

[7] Grigore Roşu and Andrei Ştefănescu. Towards a unified theory of operational and axiomatic semantics. In *ICALP*, volume 7392 of *LNCS*, pages 351–363, 2012.

[8] Grigore Roşu and Andrei Ştefănescu. Matching logic: a new program verification approach. In *ICSE (NIER track)*, pages 868–871, 2011.

[9] Andrei Stefanescu. Matchc: A matching logic reachability verifier using the K framework. In *K Workshop*, 2011. Appeared in volume 304 of ENTCS, pages 183-198, 2014.

[10] Andrei Arusoaie, Dorel Lucanu, Vald Rusu, Traian Florin Şerbănuţă, Andrei Ştefănescu, and Grigore Roşu. Language definitions as rewrite theories. In *WRLA*, volume 8663 of *LNCS*, pages 97–112, 2014.

[11] Vlad Rusu, Dorel Lucanu, Traian-Florin Şerbănuţă, Andrei Arusoaie, Andrei Ştefănescu, and Grigore Roşu. Language definitions as rewrite theories. *JLAMP*, 85(1, Part 1):98–120, Jan 2016.

[12] Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Ştefănescu. Recursive proofs for inductive tree data-structures. In *POPL*, pages 123–136. ACM, 2012.

[13] Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242. ACM, 2013.