The $\mathbb{K}$ Vision for the Future of Programming Language Design and Analysis

Xiaohong Chen[1,2(✉)] and Grigore Roşu[1,2(✉)]

[1] University of Illinois at Urbana-Champaign, Champaign, USA
{xc3,grosu}@illinois.edu
[2] Runtime Verification Inc., Urbana, USA

Abstract. Formal programming language semantics should be a unique opportunity to give birth to a better language, not a cumbersome post-mortem activity. Moreover, language implementations and analysis tools should be automatically generated from the formal semantics in a correct-by-construction manner, at no additional cost. In this paper, we discuss how we are pursuing this vision of programming language design and analysis within the context of the $\mathbb{K}$ framework (http://kframework.org), where it is easy and fun to design and deploy new programming languages; where language designers can focus on the desired features and not worry about their implementation; and where the correctness of all auto-generated language implementations and tools is guaranteed on a case-by-case basis, and every individual task, be it parsing, execution, verification, or anything else, is endorsed by its own proof object that can be independently checked by third-party proof checkers, making no compromise to safety or correctness.

Keywords: $\mathbb{K}$ framework · Programming language design · Language frameworks

1 Background

A formal semantics of a programming language is a precise, rigorous, and non-ambiguous mathematical definition of the behaviors of all programs of that language. Consider syntax first. A formal syntax of a language is a precise, rigorous, and non-ambiguous mathematical definition of which sequences of characters construct a well-formed program of that language. Scientists and engineers have found and converged on ways to write formal syntax definitions of programming languages, where regular expressions are often used to define the lexical structures and Backus-Naur form (BNF) grammars define the grammatical structures. Additionally, they developed automatic tools, like Yacc [11], that take the syntax definition and generate syntax tools such as lexers and parsers, which are specific to that language. Undoubtedly, these automatic tools greatly reduce the amount of work in designing and developing new programming languages.

E. Bartocci et al. (Eds.): Havelund Festschrift, LNCS 13065, pp. 3–9, 2021.
https://doi.org/10.1007/978-3-030-87348-6_1

But why stop at syntax? Why not do the same for language *semantics*, too? This is a question that the second author and Klaus Havelund have reflected upon numerous times during their meetings and walks as colleagues at NASA Ames during the years 2000 and 2002, to a point where the second author found it so fascinating as a question that answering it has became his main scientific goal. The second author is thus grateful to Klaus Havelund not only for introducing him to and a life-time collaboration on the topic we call today "runtime verification", but also for shaping his belief that formal semantics can and should be accessible and practical. But this was not going to be easy. Many great logicians and computer scientists have been conducting research for decades on formal semantics of programming languages, though. Since the 1960s, various semantics notions and styles have been proposed, including Floyd-Hoare axiomatic semantics [6,10], Scott-Strachey denotational semantics [17], and various types of operational semantics [1,12,13,15].

Unfortunately, it turned out that semantics is much harder than syntax. After nearly 50 years of research, semantics-based tools are still far from syntax-based tools in terms of scalability, usability, robustness, popularity, and reusability across different languages. Worse, practitioners tend to think that formal semantics of real programming languages are hard to define, difficult to understand, and ultimately useless. In practice, many language designers simply forgo defining a formal semantics for their language altogether, and just manually implement adhoc interpreters, compilers, or whatever tools are desired or needed, with little or no correctness guarantees. As a result, programs in such languages may end up manifesting unexpected behaviors after they are deployed, sometimes with catastrophic consequences.

2 The Vision of an Ideal Language Framework

We can change the status quo and make programming language design faster, easier, fun, economical, and mathematically rigorous, by using an *ideal language framework* that incentivizes programming language designers to design and implement their languages by defining a formal semantics and nothing else. All the above-mentioned language tools for their languages will be automatically generated as a bonus, as illustrated in Fig. 1. For existing languages, we are also better off defining a post-mortem formal semantics that yields the language tools following a principled, correct-by-construction way. Our main message is the following.

> It is *hard* to define a complete formal semantics of any real-world programming language, but if the objective is to tolerate no imprecision or ambiguity in the language, there is no way around it. But we can do it *once and for all*. Nothing else should be needed besides the canonical formal semantics, from which all implementations, tools, and documentations are auto-generated by the framework.

Based on our experience and thinking, an ideal language framework—*the* framework—is distinguished by the following characteristics.

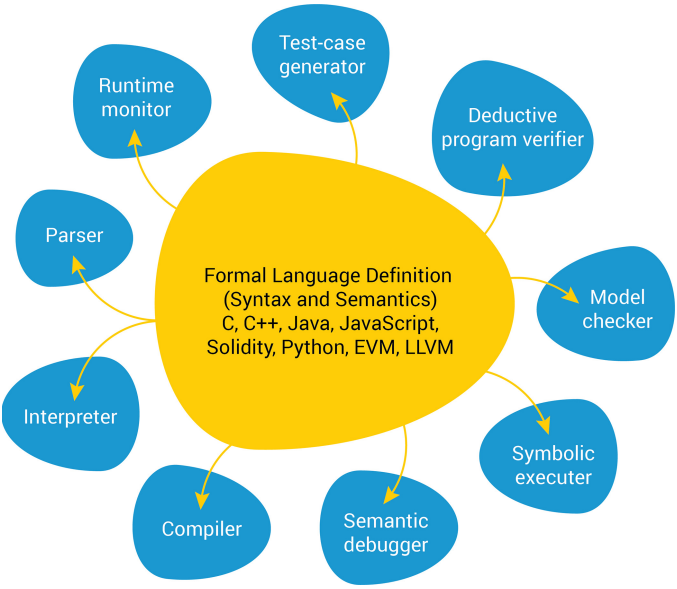

Fig. 1. 𝕂 vision: an ideal language framework

The framework should have an intuitive and user-friendly front-end interface (i.e., a meta-language) to define the formal language semantics. Language designers can use the framework as a natural means to create and experiment with their languages. Since the basic language tools like the parser and the interpreter are auto-generated, the semantics can be executed and easily tested while it is designed, simply by running lots of test programs using the semantics and check the results.

The framework should make language semantic definitions modular and extensible. Language features are loosely coupled, and language designers can easily add new features without revisiting existing definitions. This is because the modern programming languages in emerging areas such as blockchains and smart contracts are constantly evolving, with a rapid development cycle where a new release is deployed on a weekly basis.

The framework should have a solid logical foundation. The semantic definition should have a clear mathematical meaning so it can serve a basis for formal reasoning about programs. The underlying *core logic* of the framework should be highly expressive so arbitrarily complex language features can be formalized in reasonably amount of effort. Auto-generated language tools are (1) correct-by-construction, so there is no "modeling gap" between the formal semantics and the actual tools, and (2) efficient, so there is no need to waste time in handcrafting language-specific tools.

Finally, the framework should have a minimal *trust base* that is fully comprehensible and accessible to all its users. The complexity of the framework

implies that it is practically impossible to give it a perfect, bug-free implementation, which can have tens of thousands of lines of code. Therefore, instead of aiming at the correctness of the entire framework implementation, we generate *proof objects* for every individual task that the framework does, such as parsing a piece of code, executing a program snippet, verifying a formal property. These proof objects encode formal proofs of the underlying core logic and become correctness certificates that can be proof-checked by independent proof checkers. The framework users need not trust any particular implementations of the framework. Instead, correctness is established on a case-by-case basis for each individual task, each endorsed by its corresponding proof object.

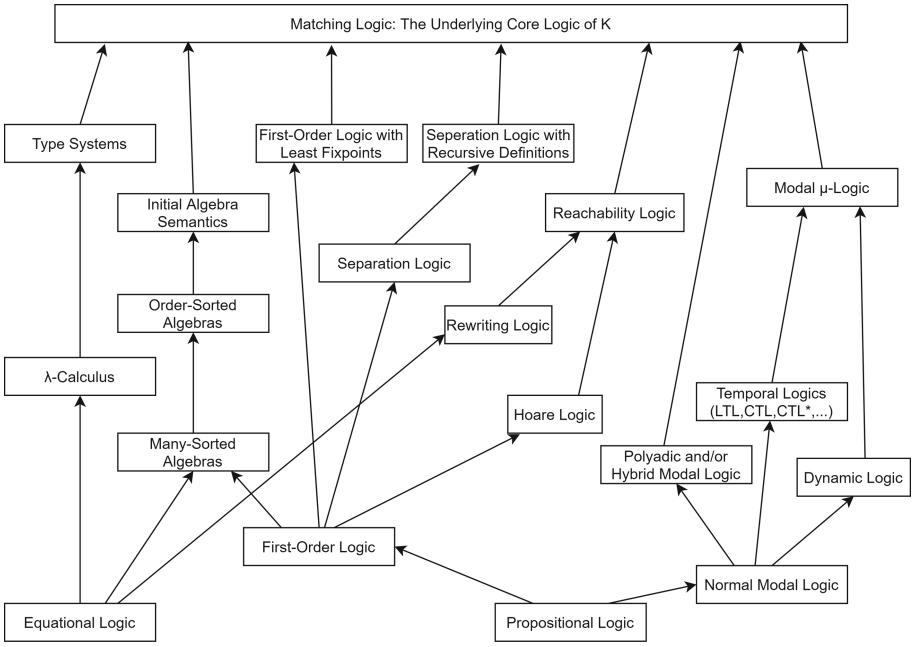

Fig. 2. The logical foundation of $\mathbb{K}$ is matching logic; it captures various logical systems and/or semantic approaches as its theories.

3 The $\mathbb{K}$ Language Framework

$\mathbb{K}$ (http://kframework.org) is a 20-year continuous effort in pushing towards the ideal vision as illustrated in Fig. 1. We have used $\mathbb{K}$ to define the complete, executable semantics of many real-world languages and automatically generate implementations and analysis tools for those languages, including C [8], Java [2], JavaScript [14], Python [7], Ethereum virtual machine bytecode [9], and x86-64 [5]. The research study and practical applications of $\mathbb{K}$ prove that the ideal scenario is within our reach to achieve in the short term.

$\mathbb{K}$ provides an intuitive and user-friendly meta-language to define the formal semantics of programming languages, based on configurations, computations, and rewrite rules. For example, the following $\mathbb{K}$ code[1] defines the formal semantics of variable lookup in a simple imperative language that has a C-like syntax:

```
rule <k> X:Id => V ...</k>
     <env>... X |-> L ...</env>
     <store>... L |-> V ...</store>
```

where `<k> ... </k>` is a configuration cell that holds the current computation. `<env> ... </env>` holds a mapping from variables to locations and `<store> ... </store>` holds a mapping from locations to values, respectively. The complete configuration can include many other user-defined, possibly nested cells that include all the information needed to execute programs, but only those which are relevant need to be mentioned in the semantic rule. Intuitively, the rule states that if the program variable `X` is mapped to location `L`, and `L` is mapped to value `V`, then rewrite `X` to `V` in the current computation, and do not change the rest of the computation, environment, or store. $\mathbb{K}$ uses the three dots "..." to denote the computation frames.

$\mathbb{K}$ definitions are highly modular and extensible. Not only that language semantics can be organized into different $\mathbb{K}$ modules, semantic definitions themselves are extension-friendly. For example, the above variable lookup rule only mentions the configuration cells that are relevant to variable lookup, and does not require to write the irrelevant computation frames. Therefore, we can extend or modify the language however we like and do not need to revisit and update the variable lookup rule, as long as we do not change how environments and stores are represented in the semantics. Language features that are loosely-coupled have their semantic definitions also loosely-coupled in $\mathbb{K}$, giving users (nearly) the maximum modularity and extensibility.

$\mathbb{K}$ has a solid logical foundation called *matching logic* [3,4,16], whose formulas are called patterns and the key concept is that of pattern matching, meaning that the semantics of a pattern is the set of elements that match it. Matching logic is highly expressive and is able to define, as logical theories, all the common logical formalisms that semanticists use to formalize languages; see Fig. 2 for some of them. Every $\mathbb{K}$ language definition yields a logical theory of matching logic, and all language tools auto-generated by $\mathbb{K}$ are best-effort implementations of heuristics of proof search within the matching logic theories.

Matching logic follows a minimalism design, requiring only the simplest building blocks, from which more complex and heavier concepts are defined using axioms, notations, and theories. One major advantage of matching logic is that it has a small Hilbert-style proof system with only a few proof rules, and checking Hilbert-style proofs, i.e., sequences of patterns $\varphi_1, \ldots, \varphi_n$, is very (!) simple. One only needs to check that every φ_i is either an axiom of the proof system,

[1] The code is extracted from the complete $\mathbb{K}$ definition of an academic language named IMP^{++}; see https://github.com/kframework/k/blob/master/k-distribution/tutorial/1_k/4_imp++/lesson_8/imp.md.

or is obtained from a proof rule. A proof object simply encodes a Hilbert-style proof $\varphi_1, \ldots, \varphi_n$, decorated with proof annotations that state how each φ_i is proved, so it contains all necessary details to be easily checked by third-party checkers.

4 Conclusion

We are witnessing a bright future, where designing and implementing programming languages can be easy, fun, well-principled, mathematically rigorous, and accessible to non-expert users. This future is made possible by holding the vision of an ideal language framework, where programming languages must have formal semantics, and all language implementations, tools, documentations, etc. are automatically generated from the formal semantics by the framework in a correct-by-construction manner, at no additional costs. We have been pursuing this ideal vision with the $\mathbb{K}$ framework and we believe with strong evidence that the ideal vision is feasible within our reach in the near future. We warmly thank our visionary friend Klaus Havelund for various discussions on this topic and for countless encouragements along the way.

References

1. Berry, G., Boudol, G.: The chemical abstract machine. Theor. Comput. Sci. **96**(1), 217–248 (1992). https://doi.org/10.1016/0304-3975(92)90185-I
2. Bogdanas, D., Roşu, G.: K-Java: a complete semantics of Java. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, pp. 445–456. ACM, New York (2015). https://doi.org/10.1145/2676726.2676982
3. Chen, X., Roşu, G.: Matching μ-logic. In: Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019), Vancouver, BC, Canada, pp. 1–13. IEEE (2019)
4. Chen, X., Roşu, G.: A general approach to define binders using matching logic. Technical report, University of Illinois at Urbana-Champaign (2020). http://hdl.handle.net/2142/106608
5. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: A complete formal semantics of x86-64 user-level instruction set architecture. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019), pp. 1133–1148. ACM, June 2019. https://doi.org/10.1145/3314221.3314601
6. Floyd, R.W.: Assigning meaning to programs. In: Symposium on Applied Mathematics, vol. 19, pp. 19–32 (1967)
7. Guth, D.: A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, August 2013. http://hdl.handle.net/2142/45275
8. Hathhorn, C., Ellison, C., Roşu, G.: Defining the undefinedness of C. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 336–345. ACM, New York (2015). https://doi.org/10.1145/2737924.2737979

9. Hildenbrandt, E., et al.: KEVM: a complete semantics of the Ethereum virtual machine. In: Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF 2018). IEEE (2018). http://jellopaper.org
10. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
11. Johnson, S.C.: Yacc: Yet another compiler-compiler (1975). http://dinosaur.compilertools.net/yacc/
12. Kahn, G.: Natural semantics. In: Brandenburg, F.J., Vidal-Naquet, G., Wirsing, M. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987). https://doi.org/10.1007/BFb0039592
13. Mosses, P.D.: Modular structural operational semantics. J. Log. Algebraic Program. **60–61**, 195–228 (2004)
14. Park, D., Stefănescu, A., Roşu, G.: KJS: a complete formal semantics of JavaScript. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 346–356. ACM, New York (2015). https://doi.org/10.1145/2737924.2737991
15. Plotkin, G.: A structural approach to operational semantics. J. Log. Algebraic Program. **60–61**, 17–139 (2004)
16. Roşu, G.: Matching logic. Log. Methods Comput. Sci. **13**(4), 1–61 (2017)
17. Scott, D.S.: Domains for denotational semantics. In: Nielsen, M., Schmidt, E.M. (eds.) ICALP 1982. LNCS, vol. 140, pp. 577–610. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0012801