Formal Design, Implementation and Verification of Blockchain Languages

Grigore Rosu
University of Illinois at Urbana-Champaign and Runtime Verification, Inc., USA
http://fsl.cs.illinois.edu/grosu
grosu@illinois.edu

Abstract

This invited paper describes recent, ongoing and planned work on the use of the rewrite-based semantic framework $\mathbb{K}$ to formally design, implement and verify blockchain languages and virtual machines. Both academic and commercial endeavors are discussed, as well as thoughts and directions for future research and development.

2012 ACM Subject Classification Security and privacy → Logic and verification, Software and its engineering → Software verification, Theory of computation → Logic and verification

Keywords and phrases Formal semantics Program verification Blockchain

Digital Object Identifier 10.4230/LIPIcs.FSCD.2018.2

Category Invited Talk

Related Version https://www.ideals.illinois.edu/handle/2142/97207

Funding NSF CCF-1421575; NSF SBIR-II-1660186; IOHK grant; Ethereum Foundation grant.

1 Introduction and Motivation

Many of the recent expensive cryptocurrency bugs and exploits are due to flaws or weaknesses of the underlying blockchain programming languages or virtual machines [6, 4, 1, 3, 12]. The usual post-mortem approach to formal language semantics and verification, where the language is firstly implemented and used in production for many years before a need for formal semantics and verification tools naturally arises, does simply not work anymore. New blockchain languages or virtual machines are proposed at an alarming rate, followed by new versions of them every few weeks, sometimes every few days, together with programs (a.k.a. smart contracts) in these languages that are responsible for financial transactions totaling more than \$1B/day only on the Ethereum blockchain [7]. Formal analysis and verification tools for such languages and virtual machines are therefore needed *immediately*.

In order to formally verify a program in any given language, a formal model of the program is necessary. Such a program model can be developed manually, in mechanical theorem provers such as Coq [10] or Isabelle [13], but this is usually expensive and thus done rarely, mostly in the context of mission critical systems and in combination with other activities, such as defining model abstractions and protocol/algorithm/model validation. The norm is for tools to extract such program models automatically, based either on translations of the program to particular intermediate languages such as Boogie [2] or Why [8] that serve as input to specialized program verifiers, or on direct implementations of Hoare logics or verification condition (VC) generators for the target programming language.

The translation approach has the advantage that the same verifier can be used across various target languages, but it has the drawback that program behaviors may be lost in

3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018).
Editor: Hélène Kirchner; Article No. 2; pp. 2:1–2:6

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

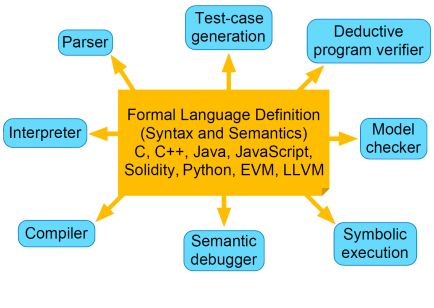
translation, so a trusted formal semantics of the original language and a proof of correctness of the translation are needed for increased confidence. Additionally, backwards translations of failed proofs need to be engineered, so users see error messages specific to their original language and not to the level of the intermediate language. The direct approach avoids both the translation correctness and the translation of failed proofs problems, but it is usually significantly more complex to implement, which can lead to subtle implementation errors that are hard or impossible to expose by testing (Hoare logics are not easily executable). Therefore, to increase confidence in such direct program verifiers, the underlying Hoare logic or VC generation procedure needs to be proved sound with respect to some trusted reference model of the target programming language, typically an operational semantics.

Therefore, a formal trusted semantics of the target programming language is required for increased confidence of program verification. The $\mathbb{K}$ framework [11] (`http://kframework.org`) takes the firm position that a formal language semantics should be needed to validate not only program verifiers, but essentially all the target language tools. Moreover and more importantly, that no other formal or informal, direct or indirect semantics of the target language should be required for any of the tools, and that the tools should be either generated automatically from or take as input the formal language semantics. That is, that all the language-specific tools for a given language should be produced automatically by the framework, correct-by-construction, from the formal semantics of the language. Figure 1 depicts the $\mathbb{K}$ belief and approach. This is nevertheless the best we can hope for in our field. But does it really work? Isn't it too idealistic? Aren't the tools too inefficient to be practical?

Some initial practical instances of the $\mathbb{K}$ approach were reported in [5], where existing formal semantics of C, Java and JavaScript were used as inputs to $\mathbb{K}$'s language-parametric program verifier, to yield program verifiers specific to these three languages. The resulting program verifiers were comparable in performance with existing state-of-the-art verifiers developed specifically for these languages. Here we bring additional evidence for the feasibility of the $\mathbb{K}$ approach, this time in the context of the blockchain. Specifically, we discuss recent academic and commercial results in designing blockchain languages and virtual machines by formalizing their semantics. Implementations for these are generated automatically from their semantics, in a correct-by-construction fashion, and so are program verifiers for them.

Why target the blockchain as an application domain for the $\mathbb{K}$ approach? First, because it is a new field in desperate need of formal verification; if cryptocurrencies are the future of money, then we ought to do our best to increase the security, safety and reliability of blockchain transactions. Second, because the entire blockchain space is a moving target, with paradigms and languages that change on a daily basis with no time to develop program verifiers following the traditional Hoare logic or VC generation approaches; therefore, it is a sweet spot for our language-parametric approach. Third, because two major blockchains holding cryptocurrencies, Ethereum and Cardano, showed unreserved interest in pushing formal methods in the design and implementation of their languages, and even deploy new versions of the blockchain using technology resulting from this research initiative. Finally, because it offers an environment where a language framework like $\mathbb{K}$ can be pushed even beyond its original, already ambitious goal: it can serve as a universal language of languages, where language semantics (or more exactly their hashes) are stored on the blockchain, and then correct-by-construction compilers and interpreters for such languages are generated automatically; this way, smart contract developers can program and verify them using their favorite languages, provided that they have a formal semantics on the blockchain.

The official version will be available from July 9, 2018 at:
http://www.dagstuhl.de/dagpub/978-3-95977-077-4




```
module LAMBDA
  imports SUBSTITUTION
  syntax Val ::= Id
               | "lambda" Id "." Exp  [binder]
  syntax Exp ::= Val
               | Exp Exp         [strict, left]
               | "(" Exp ")"         [bracket]
  syntax KVariable ::= Id
  syntax KResult ::= Val

  rule (lambda X:Id . E:Exp) V:Val => E[V / X]

  syntax Val ::= Int | Bool
  syntax Exp ::= Exp "*" Exp    [strict, left]
               | Exp "/" Exp         [strict]
               > Exp "+" Exp    [strict, left]
               > Exp "<=" Exp        [strict]
  rule I1 * I2 => I1 *Int I2
  rule I1 / I2 => I1 /Int I2
  rule I1 + I2 => I1 +Int I2
  rule I1 <= I2 => I1 <=Int I2
```

```
  syntax Exp ::= "if" Exp "then" Exp
                 "else" Exp  [strict(1)]
  rule if true  then E else _ => E
  rule if false then _ else E => E

  syntax Exp ::= "let" Id "=" Exp "in" Exp
  rule let X=E in E' => (lambda X . E') E

  syntax Exp ::=
        "letrec" Id Id "=" Exp "in" Exp
      | "mu" Id "." Exp  [binder]
  rule letrec F:Id X = E in E'
    => let F = mu F . lambda X . E in E'
  rule mu X . E => E[(mu X . E) / X]

  syntax Exp ::= "callcc" Exp  [strict]
  syntax Val ::= cc(K)
  rule <k> (callcc V:Val => V cc(K))
           -> K </k>
  rule <k> cc(K) V -> _ =>  V -> K </k>
endmodule
```

Figure 1 Left: the $\mathbb{K}$ framework approach to language design, implementation and verification. Center and right: the $\mathbb{K}$ definition of a call-by-value lambda calculus with arithmetic and callcc.

2 $\mathbb{K}$ Framework

$\mathbb{K}$ is a rewrite-based executable semantic framework in which programming languages, type systems and formal analysis tools can be defined using *configurations*, *computations* and *rules*. Configurations organize the state in units called cells, which are labeled and can be nested. Computations carry computational meaning as special nested list structures sequentializing computational tasks, such as fragments of program. Computations extend the original language abstract syntax. K (rewrite) rules make it explicit which parts of the term they read-only, write-only, read-write, or do not care about. This makes K suitable for defining truly concurrent languages even in the presence of sharing. Computations are like any other terms in a rewriting environment: they can be matched, moved from one place to another, modified, or deleted. This makes K suitable for defining control-intensive features such as abrupt termination, exceptions or call/cc. Figure 1 left depicts the $\mathbb{K}$ architecture.

Figure 1 center and right shows the complete $\mathbb{K}$ definition of a simple call-by-value lambda calculus language with builtin arithmetic, conditional, let, letrec, and call/cc. Note that syntax is define using conventional BNF, with terminals in quotes. The `|` separates production of same precedence, while `>` states that the previous productions bind tighter than the subsequent ones. A parser is generated automatically and is used to parse both the programs and the semantic rules; i.e., rules can use concrete syntax. Syntax and rule declarations can be tagged with attributes. Some attributes have meaning for the parser, such as `left` for left associativity, others have semantic meaning, such as `binder` (used by the builtin variable-capture free substitution) and `strict` (which defines appropriate evaluation contexts). For $\mathbb{K}$'s internal substitution to work out of the box, we also need to tell it which syntactic categories act as variables, by subsorting them to `KVariable`. Similarly, for efficiency we need to tell it which categories build non-reducible results by subsorting to `KResult`. Most of the semantic rules are self-explanatory. The call/cc rules use $\mathbb{K}$'s specific *local rewriting*: rewriting takes place in context, specifically in the `<k/>` cell, and not at the top level. This gives $\mathbb{K}$ additional convenience and modularity in language definitions.

Taking such formal language definitions as input, $\mathbb{K}$ generates a variety of tools for the defined language as shown in Figure 1, without any other piece of knowledge about the given language except its formal syntax and semantics. Complete languages semantics for real-world languages like C, Java and JavaScript have been defined this way, and tools for them have been generated and shown to have acceptable performance when compared to existing adhoc tools for the same languages [5].

The official version will be available from July 9, 2018 at:
http://www.dagstuhl.de/dagpub/978-3-95977-077-4

3 Current Progress

KEVM: Ethereum, the second largest blockchain cryptocurrency after Bitcoin, implements a general-purpose replicated "world computer" that allows for the development of arbitrary programs, called "smart contracts", that execute in blockchain transactions using the blockchain to synchronize their state globally. Smart contracts are written in various high-level languages, but are ultimately translated to a low-level language called the Ethereum Virtual Machine (EVM) [14]. Among other features, these contracts can tally user votes, communicate with other contracts, store or represent digital assets, and send or receive money in cryptocurrencies, without requiring trust in any third party to faithfully execute the contract. Their correct and secure operation relies entirely on the correctness of their EVM code. Any code error can be immediately exploited resulting in significant financial loss [6, 4, 1, 3, 12].

To enable the formal verification of smart contracts, in a project partially funded by the research and engineering company IOHK (`http://iohk.io`, the creators of the Cardano blockchain and the ADA cryptocurrency), we have formalized the semantics of the EVM [9]. Our $\mathbb{K}$ semantics of the EVM, which we refer to as KEVM, is as *complete* as it can be. We know this because we *tested* it by running the automatically generated interpreter (see Figure 1) against the comprehensive 40,000-program test suite that comes with the official C++ implementation of the EVM, which serves as a conformance suite for EVM implementations. Building upon KEVM, the startup Runtime Verification has formally verified several smart contracts as part of their commercial verification services (`https://runtimeverification.com/smartcontract/`).

A pleasant surprise was that the EVM interpreter automatically generated from KEVM turned out to be only *one order of magnitude* slower on average than the official C++ implementation offered by the Ethereum Foundation [9]. Since smart contracts are small and fast executing programs, the above suggests that KEVM can serve not only as a reference executable model of the EVM, but also as an actual production implementation. We are grateful to IOHK for launching a testnet on Cardano in Summer 2018 to test this hypothesis in a real-world setting. If successful, this experiment can be the first step towards a world where virtual machines are generated automatically from their formal specifications, correct-by-construction. If performance is not a problem, why should it be any other way?

IELE: One of the major lessons we learned during the EVM formalization effort was that EVM can be improved along various dimensions, improvements that could make both implementations and smart contract verification easier and faster. Instead of doing so, we preferred to design and implement a new virtual machine, IELE: `https://github.com/runtimeverification/iele-semantics`. Unlike the EVM, which is a stack-based machine, IELE is a register-based machine, like LLVM. IELE also directly supports functions, like LLVM, and is human readable. It has an unbounded number of registers and also supports unbounded integers. Like KEVM, the design of IELE was also done in a semantics-based style, using $\mathbb{K}$, and a VM was automatically generated from its formal specification. To our knowledge, IELE is the first VM that was completely designed and implemented using formal methods. There is no line of low-level code written by humans; all code is automatically generated from its formal specification. The IELE project was funded by IOHK, with the explicit objective of eventually powering real-world blockchains, including Cardano. The project is complete and like with KEVM, a test net will be deployed in Summer 2018 by IOHK to evaluate IELE in a real-world setting. To make migration of existing Ethereum smart contracts to IELE possible, we have also developed a IELE compiler for the most commonly used smart contract language, Solidity: `https://github.com/runtimeverification/solidity`.

We are currently also working on a IELE compiler for Plutus, a high-order functional programming language for future smart contracts developed by IOHK under the supervision of Philip Wadler: `https://github.com/kframework/plutus-core-semantics`.

Vyper, Casper: Vyper (`https://github.com/ethereum/vyper`) is a novel programming language for smart contracts that aims for increased security, simplicity, and human readability; Vyper currently compiles to the EVM. Casper (`https://github.com/ethereum/casper`) is a novel consensus protocol implemented in Vyper, meant to save wasteful electricity expenditures and at the same time provide greatly increased security. Vyper and Casper are both proposed by the Ethereum Foundation and, unfortunately, both are moving targets. Inconsistencies and bugs are found and fixed in Vyper on a weekly basis, which may potentially influence the Casper implementation, which itself still changes due to other forces. Funded by Ethereum Foundation, we have formalized the semantics of Vyper in $\mathbb{K}$, discovering several bugs and inconsistencies in the process: `https://github.com/kframework/vyper-semantics`; and we are also formally verifying the Casper code in Vyper, as compiled to EVM: `https://github.com/runtimeverification/verified-smart-contracts/tree/master/casper`. For verification purposes, we are currently regarding Casper as a smart contract eventually executed on the EVM, but its inherent complexity makes it highly non-trivial to correctly specify its intended behavior. To reduce the risk of misspecifying its Vyper code correctness, in a joint effort with the Etherum Foundation and the University of Texas at Austin we are also formalizing the actual protocol in Coq and in Isabelle, and validate the model by proving its intended safety and liveness properties. Then we will show that the proved Vyper code properties are consistent with the Coq and Isabelle models. All these are necessary due to the extremely important role that Casper will play in the near future for Ethereum.

4 Conclusion and Future Work

While $\mathbb{K}$ may not be the final answer to our quest for an ideal language framework, we believe that it has demonstrated that it is possible, and feasible, to generate a variety of formal execution and analysis tools for a given language from the formal semantics of that language. Moreover, only one, executable semantics for any given language suffices in order to generate all the tools, and that the so generated tools can be correct-by-construction, thus eliminating the need for redundant semantics and complex proofs of correctness.

Some years may still need to pass before sufficient evidence is accumulated to convince the skeptical formal methodist that the approach has merit even with mainstream languages like C and Java, for which well-engineered formal verification tools already exist. But for emerging fields like the blockchain, which come with new languages that routinely change every few days and require mostly small but tricky programs, the language-parametric semantic framework approach appears to be the only solution quickly available.

We hope that this wave of interest in language frameworks like $\mathbb{K}$ will lead to the development of several important advances in the field, which will then be applicable across all languages. On the foundational side, we need to develop language-/paradigm-independent logics that allow us to specify any desired properties about any programs in any programming languages. On the practical side, automation is critical for the success of any verification environment. Also, generation of proof objects to act as correctness certificates for the various formal tools generated for a given language would increase demand and adoption of such tools, especially in the blockchain domain.

Acknowledgments: This work would have not been possible without the sustained dedication of the $\mathbb{K}$-team (`http://www.kframework.org/index.php/People`) and numerous

other contributors and enthusiasts. I would like to particularly thank Philip Daian, Everett Hildenbrandt, and Charles Hoskinson for bringing the blockchain needs for formal verification to our team's attention, and for evangelizing our language-parametric verification approach in blockchain communities. Warm thanks to Hélène Kirchner and the entire FSCD'18 program committee for inviting me to present this work at the conference and to submit this paper.

References

1 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts. *IACR Cryptology ePrint Archive*, 2016:1007, 2016.

2 Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 364–387, 2006.

3 Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An in-depth look at the parity multisig bug. 2017. `http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/`.

4 Vitalik Buterin. Thinking about smart contract security. 2016. `https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/`.

5 Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, Nov 2016.

6 Phil Daian. DAO attack, 2016. `http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`.

7 Etherscan. Ethereum transactions, 2018. `https://etherscan.io/`.

8 Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177, 2007.

9 Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KEVM: A Complete Semantics of the Ethereum Virtual Machine. In *Computer Security Foundations Symposium (CSF'18)*, 2018. `http://jellopaper.org`.

10 The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0. URL: `http://coq.inria.fr`.

11 Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

12 Jutta Steiner. Security is a process: A postmortem on the parity multi-sig library self-destruct, 2017. `https://blog.ethcore.io/security-is-a-process-a-postmortem-on-the-parity-multi-sig-library-self-destruct/`.

13 The Isabelle development team. Isabelle, 2018. `https://isabelle.in.tum.de/`.

14 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014. (Updated for EIP-150 in 2017) `http://yellowpaper.io/`.