CHALMERS
UNIVERSITY OF TECHNOLOGY

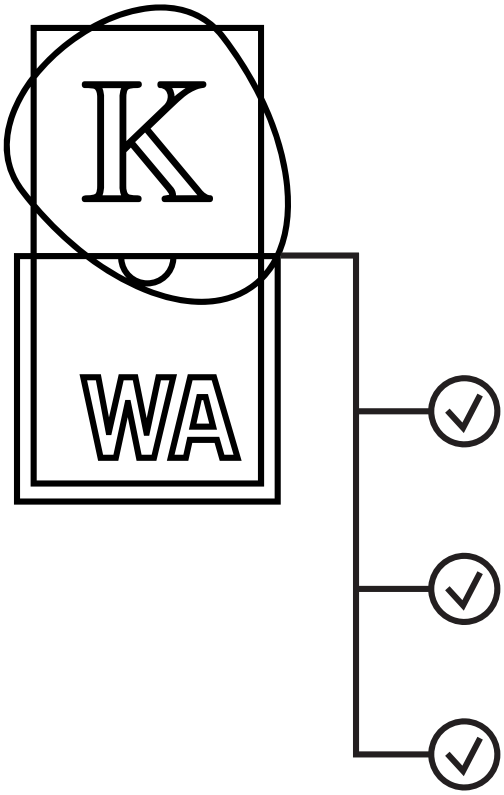
UNIVERSITY OF GOTHENBURG

Formally Verifying WebAssembly with KWasm

Towards an Automated Prover for Wasm Smart Contracts

Master's thesis in Computer Science and Engineering

RIKARD HJORT



Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Formally Verifying WebAssembly with KWasm

Towards an Automated Prover for Wasm Smart Contracts

RIKARD HJORT

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Formally Verifying WebAssembly with KWasm
Towards an Automated Prover for Wasm Smart Contracts
RIKARD HJORT

Supervisor: Thomas Sewell, Department of Computer Science and Engineering
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Conceptual rendering of the KWasm system, as the logos for $\mathbb{K}$ ans WebAssembly merged together, with a symbolic execution proof tree protruding. The cover image is made by Bogdan Stanciu, with permission. The WebAssembly logo made by Carlos Baraza and is licensed under Creative Commons License CC0. The $\mathbb{K}$ logo is property of Runtime Verification, Inc., with permission.

Typeset in LaTeX
Gothenburg, Sweden 2020

Formally Verifying WebAssembly with KWasm
Towards an Automated Prover for Wasm Smart Contracts
Rikard Hjort
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

A smart contract is immutable, public bytecode which handles valuable assets. This makes it a prime target for formal methods. WebAssembly (Wasm) is emerging as bytecode format for smart contracts. KWasm is a prototype mechanization of Wasm in the K framework which can be used for formal verification. The current project aims to verify a single Wasm smart contract, a simple token contract. First, we complete the KWasm semantics to be able to deal with modules and test the semantics against the official conformance tests. This reveals several shortcomings of the conformance tests. Second, we create an Ethereum embedding by creating a separate Ethereum client semantics and combining it with the KWasm semantics through a thin, synchronizing interface. This embedding is then unit tested, including conformance tests for two variants of the WRC20 contracts. Thirdly, we use the KWasm semantics to prove properties of incrementally more complex Wasm programs, with an emphasis on heap reasoning, culminating in the verification of a helper function that reverses the bytes of a 64-bit integer. In the process we add 25 axioms that get upstreamed into K, and several more that are useful for general Wasm verification.

Keywords: K, K framework, WebAssembly, Wasm, Ethereum, Ewasm, formal methods, formal verification, semantics, specification

Acknowledgements

I want to express my sincerest thank you to Everett Hildenbrandt, who started the KWasm project, for letting me contribute, for teaching me about $\mathbb{K}$ and all its quirks and for supporting me with design discussions and troubleshooting in my day-to-day work.

A big thank you also to Thomas Sewell for supervising my thesis project and helping me turn my engineering work into useful academic work, as well as providing an inquisitive outside perspective. Thank you also to Wolfgang Ahrendt, the examiner for this thesis, for your feedback during the process and providing your insights on how this projects fits inside research field of formal methods.

Thank you Martin Lundfall for initially suggesting I should familiarize myself with KWasm, and Magnus Myreen for pushing me into pursuing it as research work. To Jakob Larsson for feedback and helping me fine tuning the thesis.

Thank you to the other contributors to KWasm, Stephen Skeirik and Qianyang Peng, for fruitful discussions and enjoyable collaboration. To Runtime Verification, Inc—especially Bogdan, Grigore, and Patrick—for giving me a shot at making this a real product.

Thank you, all.

Rikard Hjort, Gothenburg, February 2020

Contents

List of Figures

List of Code Listings

1
Background

1.1 Introduction

WebAssembly (Wasm) is a platform-independent bytecode[17]. It is supported by all major web browsers and has standalone implementations. Wasm is a low-level stack-based language with many higher-level features, including static typing, jump-free control flow, and function calls and primitives. It is designed to be portable, fast to transmit and easy to map onto modern instruction set architectures.

Wasm is meant to be *embedded*, meaning a Wasm program is part of another program that calls into it. Wasm programs are not scripts, but rather a collection of functions and variables, and there is no natural empty point such as the `main` functions in other languages like C, Java or Haskell. The embedder interacts with Wasm by modifying a shared memory array that both the embedder and Wasm program can read and modify, and by invoking Wasm functions. The Wasm program can return control to the embedder by either terminating its execution, possibly returning a value or by calling *host functions* which look like regular imported functions in the Wasm program. One can thus think of Wasm programs as symbiotic with other languages, interacting via a two-way foreign function interface.

The following toy Wasm module showcases a few features of the language (we will cover these and more in-depth in Sect. 1.8). The module declares two *functions*, `$main` and `$set-and-return`, and one page (64 KiB) of *memory*. The first function calls the second, and the second sets the value of eight bytes of memory, returning the old memory contents. In this case, we store the eight bytes representing the value `10` at memory location `42`. The functions each declare what *parameters* they expect, what their *return type* is, and if they use any *local* variables apart from the parameters. The rest of each function is the body, which is a sequence of *instructions*.

Listing 1.1: Introductory Wasm example.

```
1  (func $main (result i64)
2    i32.const 42          ;; Push first param  ($address)
3    i64.const 10          ;; Push second param ($value)
4    call $set-and-return ;; Pop params, use for call.
5  )
6
7  (func $set-and-return
8        (param $address i32) (param $new i64)
9        (result i64)
10       (local $old i64)
```

```
11    local.get $address    ;; Push param $address.
12    i64.load              ;; Pop $address, push i64 from memory.
13    local.set $old        ;; Pop i64, store in local $old.
14    local.get $address    ;; Push param $address.
15    local.get $new        ;; Push param $new.
16    i64.store             ;; Store $new to $address in memory.
17    local.get $old        ;; Push $old.
18  )
19
20  (memory 1)
```

Now assume that we would like to verify this program, i.e. we would like to make certain assertions about what its possible behaviors are, and under exactly which circumstances we expect each behavior. For example, if we were to call the function `$main` twice, what should we expect it to return? We do not know what the first call would return—it depends on the state of the memory—but we should expect the second call to return `10`. However, there are caveats. For example, it depends on what we mean by "calling twice". If other instructions may occur between the calls, those instructions may modify memory. If we want to prove beyond doubt that the second call will return 10, we must make clear under what conditions we expect it.

While the above example may seem trivial, correctly and formally specifying the behavior of even such a simple program comes with pitfalls, and proving that it adheres to the specification requires using a formal specification of Wasm as inference rules. A formal proof of a program requires a formal definition of the language it is written in. Luckily, Wasm has a rewrite-based formal operational semantics[31], which one could use to hand-verify the program above. But hand-verifying a program is time-consuming—for any sizable program, we would like for the reasoning to be at least mostly automated.

While programs running on the Web may not be renowned for their high-assurance properties, Wasm has use cases beyond the Web. It is a portable byte-code format that is meant to be inherently platform-agnostic, in the same spirit as the Java Virtual Machine. As such, it is finding uses beyond the browser. One example of such a use case is blockchains.

Blockchains are used for decentralized decision making, code, and money. The market cap for blockchains is counted in hundreds of billions of dollars[1].

At least four high-profile blockchain projects use Wasm: Ethereum has *Ewasm*[2] as a virtual machine specification for executing smart contracts; NEAR[3] will use their own Wasm flavor in the same way; EOSIO[4] already is using a Wasm VM; and Polkadot[5] has *Substrate*, a framework for building blockchains in a parameterized and modular fashion, that compiles to Wasm.

Smart contracts on blockchains handle valuable assets. The code is public and immutable. If a smart contract contains a bug that can be exploited for profit,

[1] As of Saturday 21st March, 2020: `https://coinmarketcap.com`

[2] `https://github.com/ewasm/design`

[3] `https://docs.nearprotocol.com/docs/roles/developer/quickstart/#smart-contract-development`

[4] `https://eos.io/news/eos-virtual-machine-a-high-performance-blockchain-webassembly-interpreter/`

[5] `https://polkadot.network/technology`

the assets are in danger of being stolen. The same is true if there is a bug in the software that manages network consensus, such as the Substrate framework. That is why, with blockchains adopting Wasm both for on-chain execution and as a client implementation language, it becomes important to establish with high confidence the functional correctness of the code.

1.2 Project Outline

This project is about verifying Wasm programs by using and extending an automated prover. We start with simple programs and incrementally increase the complexity. We have a Wasm program that implements a typical Ethereum smart contract that will guide the effort. As we verify more complex programs, we will need to add further reasoning power to the prover.

The prover is based on KWasm, the unfinished mechanization of Wasm in the $\mathbb{K}$ language. KWasm can be used as input to the $\mathbb{K}$ framework[33], which is a set of tools for automatically generating common programming language tooling from single language mechanizations. One of these tools is a deductive program verifier[36] that can be used to symbolically execute Wasm programs, and verify functional specifications.

The research questions this project tries to answer belong to two broad themes: completing the mechanization, and using it for proofs.

Related to the mechanization, we ask: *What does it take to fully mechanize the Wasm specification? How can we mechanize a blockchain embedding?*

Related to proving, we ask: *Can a symbolic execution engine prove interesting properties of Wasm? Which theorems must be added as axioms? What challenges does verifying Wasm pose?*

The first set of questions can be answered by Part 1, and the second set by Part 2 of the project:

P1: Finishing KWasm and making a prototype embedding. At the outset of this project, KWasm is still a prototype. Importantly, it lacks support for modules, exporting and importing, and embedding the Wasm semantics in an environment it can interact with through a host interface. In this part, we implement the missing functionality and run conformance tests against KWasm, and we design and implement and Ewasm embedding.

P2: Using KWasm to verify Wasm programs. We give a proof-of-concept that Wasm programs can be verified for interesting properties, and construct methods for dealing with hard-to-verify properties.

1.3 The Case for Verifying Wasm

Source-level verification of high-level language code does not give as reliable results as verifying lower-level compile targets since (unverified) compilers can introduce bugs[41] [6]. To avoid trusting a possibly buggy compiler, the verifier can work on the

[6] In the Ethereum space, this Serpent compiler bug is a documented example: `https://medium.com/@AugurProject/`

lowest available level of compiler-generated target code instead of the source code.

Exploits in generated Wasm code are already being discovered [25]. If Wasm becomes a global standard for distributed bytecode, it stands to reason that it will also be used in high-assurance systems. If the same high-assurance software is to run in multi-platform environments, such as IoT networks or the Web, the verification effort should not have to be duplicated per platform. Since the semantic gap between Wasm and executable bytecode is significantly smaller than the step from a general high-level language to Wasm, the trusted computing base (TCB) is smaller than a regular compiler if the verification happens at the Wasm level, and therefore more trustworthy. In these cases, verifying Wasm code gives a nice trade-off between effort expended and assurance level.

In the case of smart contracts, this semantic gap is no longer relevant. The bytecode semantics is what consensus is formed around on the blockchain, not how that bytecode is executed on any particular—and possibly buggy—virtual machine. Since blockchains are built on protocols rather than any particular hardware, and no single piece of hardware can cause the protocol to fail, verifying the bytecode means verifying the actual contract.

Whatever environment and embedding Wasm-based high-assurance software runs in, formal verification will need to verify pure Wasm functions, as well as the interaction between the embedder and Wasm. The exact Wasm code and what invariants are necessary to preserve will vary by use-case.

To be able to use Wasm in any application, an embedding is needed. Apart from the embedding invoking Wasm code, the Wasm code may also call out to host functions to interact with the embedder. In Ethereum, the embedding (also "environment" or "execution environment") contains blockchain state information and the Ethereum host functions (or "environment interface"). Together with the Wasm specification, this is called Ewasm[7], which is a complete virtual machine for Ethereum. Ewasm is still being designed, so there are only prototypes for us to work from.[8] This project assumes that by working from this prototype for now and trying to use it for verification, lessons for the future can be gleaned that will be useful once we implement the final version of Ewasm.

Deductive program verification, i.e. proving using formal logic that a program fulfills a specification, gives the highest possible assurance of the behavior of a piece of code. It shows beyond doubt—except any doubt in the tools that generate the proof, the process for checking the proof, or the logic—that a program will always behave according to its specification. Deductive program verification is not reliant on an abstract or incomplete model, but instead uses the formal semantics of a programming language as axioms. KWasm generates a deductive program verifier that is largely automated, making it both powerful and tractable to use.

serpent-compiler-vulnerability-rep-solidity-migration-5d91e4ae90dd

[7] https://github.com/ewasm/design

[8] The available specification is based on what is commonly called Ethereum 1.X—the current, simple protocol without sharding and proof-of-stake consensus mechanism. Ewasm is planned to be incorporated in the upcoming Ethereum 2.0, and the environment may then look much different. It is not clear, for example, how one would make calls between contracts in a sharded environment.

1.4 Related Work

It has been said that deductive source-level verification has a "killer application" in smart contracts[2]. Much of the argument applies just as well to bytecode level verification, with the exception that bytecode verification may be more inscrutable for the high-level programmer.

A complete specification of the semantics of Wasm in $\mathbb{K}$ has not been done previously but there is plenty of research surrounding the $\mathbb{K}$ framework[9]. Several languages have been fully or mostly formalized in $\mathbb{K}$, with papers in high-quality publications produced describing them: C[18, 13], Java[7], JavaScript[28] are examples of large and widely used such programming languages. The most similar effort to this project, however, is KEVM [19]. This is the $\mathbb{K}$ formalization of the Ethereum Virtual Machine (EVM). EVM bytecode is what smart contracts currently are composed of on Ethereum. The KEVM paper has a similar focus to the proposed work—it describes the semantics of the EVM specified in $\mathbb{K}$ and uses them to prove properties of EVM programs.

KEVM is being used for verifying smart contracts on the Ethereum main-net, among them contracts handling hundreds of millions of dollars[10]. There is also tooling being developed based on KEVM to help developers do verification and run their code in a trusted EVM version[11]. $\mathbb{K}$ semantics and tools thus already have some traction in the formal verification community around Ethereum.

While the $\mathbb{K}$-generated deductive prover is automatic, it needs help. Experience with KEVM[29] shows that some trusted theorems need to be introduced as axioms and language-specific abstractions are needed.

Wasm has previously been successfully mechanized in Isabelle, and the mechanization has been used to verify the soundness of Wasm's type system[38]. The mechanization has also been extended with additional types, which can be used to guarantee constant-time execution to avoid side-channel attacks on cryptographic libraries [39]. To the best of our knowledge it has not been used to verify any properties of specific programs or aid automatic proving, and it seems this is not the target of the project. $\mathbb{K}$, on the other hand, provides good tooling for such verification[36]. Of course, the Isabelle mechanization could also be used to prove properties of specific programs interactively. This does not, however, align with our goal of a mostly automated prover. While Isabelle is a popular verification tool, it is currently not being put to much use for verifying smart contracts, as far as we can tell. We believe there is no community or specific tooling around the Isabelle mechanization for verifying concrete programs, in part because smart contracts are not yet being written in Wasm. For example, the ETH-Isabelle mechanization of the EVM[12] has not seen as much traction as KEVM. By developing KWasm, we hope to make a complementary tool to the existing mechanization.

[9] `http://www.kframework.org/index.php/K_Publications`

[10] For example, multi-collateral DAI (`https://github.com/dapphub/k-dss`) and several other tokens and smart contracts verified as part of audits (`https://github.com/runtimeverification/verified-smart-contracts`).

[11] `https://runtimeverification.com/firefly/`

[12] `https://github.com/pirapira/eth-isabelle`

1.5 Project Limits

This project will be limited to using KWasm for verification, and will not consider other possible mechanizations. Furthermore, it will not be building the mechanization from scratch but instead, build upon the existing KWasm semantics. KWasm is a tool that we are familiar with, as we have already contributed to the project. We also do not believe that there is any point in surveying other tools at this point, because there are few to survey; a cursory search for Wasm verification tools reveals there is little beyond the Isabelle mechanization[38], which is aimed at proving high-level properties of the language[13].

The task at hand is to verify concrete Wasm programs with symbolic inputs. This work is exploratory and qualitative. Building infrastructure for large-scale automated reasoning is out of scope in the interest of time.

Finally, this project will not spend any significant amount of time evaluating what are the most relevant properties of specific Wasm programs to prove, or Wasm programs in general. Instead, it focuses on verifying what we believe are reasonable specifications of existing programs without arguing in-depth for the choices.

1.6 Ethical Considerations

The explicit goal of the proposed verification efforts is to verify some properties of smart contracts. This is the first project to use Wasm to do so, and as such may set a precedent. As previously mentioned, these programs may handle large sums of money, will be public and will be immutable. It is therefore important that we communicate with absolute clarity to current customers and future users of the system what a proof entails. There are three main concerns: understanding the proof claim, trusting the verification engine, and trusting the specification and axioms.

Firstly, formal verification always comes down to producing a mathematical statement: given certain assumptions, some conclusions follow. Fully understanding the assumptions and the result may require a certain knowledge of the logic that was used and of the domain the proof is over. Therefore, some of the assumptions may be plain wrong, or either the assumptions or results may be misinterpreted leading to incorrect actions.

Secondly, while proof of a property in $\mathbb{K}$ gives a very high assurance that the property truly holds, $\mathbb{K}$ does not yet produce machine-checkable proof objects. Thus, a bug in $\mathbb{K}$ could produce a faulty proof. While the risk that any given proof passed due to such a bug is small, it should not be dismissed. As such, $\mathbb{K}$ together with the SMT solver used, Z3, (see Sect. 1.7.3) make up a trusted computing base.

Thirdly, another possible source of erroneous verification is an error in the manually added axioms. As $\mathbb{K}$ cannot prove all desired properties just from the semantics, we manually write certain axioms that are not explicitly part of the language semantics. A mistake in such an axiom mean that we can prove properties that do not hold. It is thus important to take the utmost care in designing the set of ax-

[13] Available here: `https://www.isa-afp.org/entries/WebAssembly.html`

ioms and also to communicate clearly that all verification takes place within that axiomatization—a user who does not believe in any of the axioms should also not trust the proofs which make use of them.

Personal experience has shown that the term "formal verification" elicits in some people the belief that the verified software is infallible. Any behavior perceived as a bug, or any event labeled a "hack" on such software would then be a violation of that belief. As the person in question may have acted on that belief by, say, transferring money to a contract which was later compromised, the consequences of such a misunderstanding could be costly. Indeed, there is evidence that formally verified systems exhibit bugs that they have been "proved" to not have[14]. It is important that we as verifiers both make clear to ourselves and our clients what exactly we have proven, and that we make certain efforts to falsify our assumptions. Herein lies the main ethical consideration: we must not oversell our product (the verification) even in the face of high expectations and a fairly low risk to us as verifiers. We must explain that while the results are trustworthy (with the already mentioned caveats), the verifier may not be a domain expert, and the semantics we build, the axioms we add and the assumptions we make must be thoroughly checked.

Apart from the above caveats, like many software construction projects, this one comes at no considerable risk to people or property. No other projects yet depend on KWasm, and there are no significant business interests staked on it. What is being staked in this project is the developers' time. The worst-case scenario for this project is a null result, where nothing is accomplished at all, but nothing is lost.

1.7 Preliminaries

In this section we will give the requisite introduction to follow along the bulk of this thesis. Here, the reader will be familiarized with $\mathbb{K}$, Wasm, and formal verification. The treatments here are meant to only supply the minimum necessary understanding of each topic. The interested reader should look to the references for more complete treatments.

1.7.1 WebAssembly (Wasm)

Wasm[17] was introduced as a low-level language for fast and safe web programs. It is designed to execute fast on modern processors, be easy to validate in a single streaming pass, and uses a control flow semantics with no jumps and separation of code and memory, adding safety. Wasm is statically typed, supports modules with imports and exports, mutually recursive functions, indirect calls, and locally scoped variables. The semantics is defined as a transition system with only a few non-deterministic transitions. The semantics is reminiscent of a stack machine. Wasm has no exception handling.

The simple type system of Wasm allows static validation, such that no valid Wasm module can violate the memory model. The type system has been proven sound through an Isabelle mechanization[38]. Wasm is also amenable to extensions: for example, support for garbage collection, tail-call optimizations, and many other features are on the standards track. There are also extensions to the type system,

for example, to ensure information flow security and timing attack security [39]. This is one example of how researchers are taking an interest in Wasm for achieving high security of a low-level binary format.

However, the use-case for Wasm extends beyond the web: it is a platform-agnostic low-level language. Notably, it has found use in blockchain smart contract development, where the same code must run on many different computers, preferably at the least cost possible. There is much activity in this space, and notably, Ethereum is planning to use Wasm as the underlying virtual machine for smart contracts in the future[14].

Wasm is a more structured format than regular assembly language. A Wasm program is defined as a *module*, which contains any number of *definitions*. These define functions, byte-array memories, tables of functions for indirect function calls, and global variables. Wasm code is embedded in some environment, and is executed by calling into one of the functions in a module. Several modules can interact through imports and exports.

Functions contain stack-based code. The stack can contain not only operands but also the control flow constructs *labels* and *frames*, which serve similar purposes to their namesakes in regular assembly. Labels serve as branching targets, and frames as stores for local variables and for data necessary to return control to the caller. There is no way to perform arbitrary jumps—branching can only be done to the beginning of a loop or the end of the body of a block or conditional statement. Note that control flow structures like loops and conditional statements as well as functions are primitives in Wasm, unlike regular assembly code.

Execution wise, a function body is considered a single stack that is repeatedly rewritten until it only contains constant values. The stack has the shape of a cons list and grows to the left. Both operands and operators live on the same stack in the official specification (but in KWasm there is a special stack for operands). For example, the following is a stack rewrite is for a binary operator, `i32.add`, so its execution affects the top three stack elements: the operator and the two operands.

$$(i32.const\ 1)\ (i32.const\ 2)\ (i32.add) \hookrightarrow (i32.const\ 3)$$

Thus, the top three stack elements have been reduced to one element. (The rest of the stack below `i32.add` on the left-hand side and below `i32.const 3` on the right-hand side is implicit.)

Labels specify their arity, n, which is the number of constant values they return, and their continuations, which is a sequence of instructions, which are only taken when the label is branched to. The branching instruction `br` takes a number, which is the number of labels to break out of, reminiscent of de Bruijn indexing[10]. Thus, the instruction `br 0` means "break to the nearest label"—there may be more labels preceding the label being broken to[15].

$$\ldots\ \text{label}_n\{instr^*\}\ \text{val}^n\ (\text{br}\ 0)\ \text{end} \hookrightarrow \text{val}^n\ instr^*$$

When a label is on top of the stack, it is reduced to its result, which is n constants.

[14] `https://github.com/ewasm/design`, accessed 2020-01-08.

[15] The notation used here is simplified compared to that in the official specification, in order to simply explain these concepts, rather than give a complete semantic definition of labels.

$$\text{label}_n\{\_\}\ \text{val}^n\ \text{end}\ \hookrightarrow\ \text{val}^n$$

Frames behave similarly. They are processed when they are at the top of the stack so that the calling context is restored and values are returned.

The above examples use the abstract syntax of Wasm. Wasm also has two concrete syntaxes: a binary format, and a text format. The binary format is designed be compact and easy to validate in a single pass, ideal for streaming code. It imposes an exact representation of a module with little wiggle room, for example allowing only one ordering of the different declarations in a module, The text format is designed to be readable. It allows for many different syntactic sugars, and declarations in a module can come in any order.

1.7.2 Blockchains, Ethereum and Smart Contracts

The basic structure of blockchains is presented in the original Bitcoin paper[27] which serves as an excellent introduction to the field. Blockchains make distributed (eventual) consensus possible for a sequence of events[4, pp. 217-218]. These events can consist of arbitrary data—currency transactions and virtual machine executions being the most common types. A typical blockchain works by having each node in the network replicating the blockchain data. This means it is relatively costly to perform any action on a blockchain compared to regular, non-replicated systems, and that all the data stored in it is public. The cryptography, consensus mechanisms, network structure, game theory and economics of blockchains are all interesting topics in their own right. For this thesis, however, we are content to view them as platforms for ordering virtual machine executions, which is what we are aiming to verify.

When a blockchain stores virtual machine executions, it is considered a smart contract platform. Smart contracts make up a computing environment, the code, and state of which the network reaches consensus on.[37, pp. 9–12]. The execution model is that of a single, sequential and fully deterministic computer, which ensures all nodes can reach consensus of the exact state changes introduced in a block. A prominent example of a smart contract and decentralized application platform is Ethereum[16]. Ethereum uses a specially constructed bytecode language, EVM bytecode, to run the Ethereum Virtual Machine. The formal specification of EVM bytecode and the EVM is given in the Yellow Paper[40]. The contracts are executed by a user sending a *transaction*, unlocking a little bit of their balance of Ether—the native currency on Ethereum—to pay for the effects of the transaction, and possibly some additional Ether which the contract can then use, either by keeping it, sending it to someone else, or sending it back, or any combination of the two that keeps the total amount of Ether constant. Since each node in the network replicates the virtual machine and runs all program invocations, there must be a cost attached to storing data and running programs, since otherwise an attacker could at will cripple the entire network by sending large amounts of data or running non-terminating programs. In Ethereum this attack vector is mitigated by the introduction of "gas",

[16]Outlined in the White Paper: `https://github.com/ethereum/wiki/wiki/White-Paper`

which is a cost attached to each action on the network and each transaction run, paid by the sender and denominated in ether (at a rate decided by the sender). Storing data costs gas, as does running a non-terminating program—the program will run out of gas, causing the sender to lose ether while having no effect on the virtual machine, since out-of-gas exceptions cause the state of the machine to revert to what it was before the transaction.

Smart contracts are simply computer code on a special machine. In addition to being regular computing machines—e.g. stack machines—they have a primitive concept of addresses and currency. Contracts occupy an address and hold value in the form of cryptocurrency, such as Ether, and transact it as a computational primitive. The contracts are immutable, so actors can trust that "code is law"—the contract they have sent money to cannot change behavior after the fact. Smart contracts can, for example, implement additional currencies, often called tokens, by simply maintaining a mapping between addresses and balances, which can be modified through transfers. They can also call out to each other in an agent-like computer system. Such systems range from the dead simple, such as a single basic token contract, to the relatively complex, such as the multi-collateral DAI system[17].

While extensive formal verification such as deductive source code verification and symbolic execution (see Sect. 1.7.3) is often time-consuming and expensive, such verification efforts become more reasonable in the case of smart contracts. Assuming the effort required to formally verify a program is proportional to the number of lines of source code, the cost-to-benefit ratio of verifying smart contracts look promising. The argument is that while a smart contract hack may cause less damage than a space shuttle crash or errors in a self-driving car's decision procedure, the complexity and size of the codebase in a smart contract is often several orders of magnitude smaller. The Linux kernel contained over 5 million lines of code as of 2008[22]. If 1.3 trillion USD could potentially be lost due to any single error in the Linux kernel, then it would have the same potential loss of value per line of code as multi-collateral DAI does.[18] In the short history of Ethereum, several high-profile and expensive attacks have taken place, [5] perhaps the most prominent being an attack on TheDAO, in which an attacker stole in the vicinity of 60 million USD.

1.7.3 Formal Verification

Formal verification and formal methods are deep fields with many branches, including advanced deductive verification, symbolic execution, satisfiability modulo theory (SMT) solving, type theory, automata theory and model checking to name a few. In essence, it is the application of mathematical methods, especially formal logic[19] (the "formal part"), to the task of showing that computer programs fulfill certain specifications (the "verification" part). While some kinds of formal verification are commonplace—a type checker is a proof checker due to the Curry-Howard corre-

[17] `https://docs.makerdao.com`

[18] Granted, this is not an entirely fair comparison since the Linux kernel contains many modules that any given system may not make use of. Still, 5 million lines of code do not stand out as unusual for a software system in a critical application.

[19] The reader is directed to any introductory textbook on the subject, such as [21], chapters 1-2 for a primer.

spondence (see [16], in particular, Sect. 3.5 for an introduction) and many IDEs offer some static checking based on dependency graphs—others are more esoteric. This project uses *deductive verification* with the aid of SMT solving, through symbolic execution, to verify *functional correctness*. In theory, at least, these methods allow us to give full functional specifications of computer programs and prove them.

1.7.3.1 Deductive Verification

Deductive verification is any verification that proceeds like a deductive proof, for example like a proof in natural deduction or in sequent calculus[15, Sect. 2-3]. Preconditions are thus initial assumptions judged to be true, and after suitable reduction of the program, a deduction must be made that shows that the preconditions in conjunction with the reduced program imply the postconditions. Note that this means that the program itself must be expressed in the logic in which the proof is taking place. This can be done by either expressing the entire programming language in a suitable logic, by embedding it in the specification language (thereby extending that language), or by translating a program into a series of verification conditions[6, p. 7]. The expressibility of the logic dictates what properties can be stated, and whether or not it is decidable dictates whether or not all properties can be proved (or disproved) automatically. A simple example of a system for deductive verification is Hoare logic[20], in which a programming language's syntax is translated into conditions that hold before and after each step in its execution. Using Hoare logic, the desired proof that a program such as `S1; S2` satisfies postconditions `Q` given preconditions `P` is written `{P}S1; S2{Q}`. Letting semicolon play its usual role in imperative languages, this can further be translated to the proof obligations `{P}S1{R}` and `{R}S2{Q}`. `{P}S1{R}` and `{R}S2{Q}` are further broken down into further proof obligations until they can not be decomposed further. The verifier, which may be human or automatic, then needs to prove each obligation, using the logic of choice, and the consequence rule, given below, as needed:

$$\frac{P \to P' \quad \{P'\}S\{Q'\} \quad Q' \to Q}{\{P\}S\{Q\}}$$

Using a similar rule is common in all types of deductive verification, and is often referred to as "weakening the preconditions" when $P \neq P'$ and "strengthening the postconditions" when $Q \neq Q'$..

1.7.3.2 Functional Correctness

Functional correctness is the property of a system of adhering to a specification starting from any initial state and for any input, the specification being of observable behavior of the system rather than its internal structure.[6, p. 3] While "any initial state" and any output may sound impossibly restrictive—how could we prove a state is unreachable if the system could start in that state, or that a program that only accepts certain inputs can handle any input?—it is not so, since the requirement may rule out certain initial conditions and inputs. If, for example, a program which only accepts non-negative integers as inputs gets a negative integer, the specification

would have a precondition violated, letting us conclude $\bot$ and therefore also Q, since $\bot \rightarrow R$ for all propositions R.

1.7.3.3 Satisfiability Modulo Theory (SMT) and Their Solvers

In Boolean satisfiability (SAT) problems the goal is to find whether there is an assignment of truth values to propositions $P_1, ..., P_n$ such that a Boolean formula containing only these propositions evaluates to *true*. This has turned out to be a particular kind of problem for which there are very efficient tools, although the problem is NP-complete in general. An SMT problem is a SAT problem where the propositions may additionally hold meaning in some theory, such as the theory of Peano arithmetic, and the goal is instead to find whether values can be assigned to all free variables in the formula. The goal is to do almost as well as for SAT problems, but for a broader class of problems. Thus, an assignment of truth values that solves the corresponding (weaker) SAT problem may not solve the SMT problem. For example, the SMT problem $x < 2 \wedge x = 4$ can be solved by turning it into the corresponding SAT problem $P_1 \wedge P_2$. Since the arithmetical formulas are unequal, they can be considered different propositions. The SAT version is satisfiable with $P_1 = P_2 = true$, while the SMT problem is unsatisfiable since there is no assignment of x under which both parts of the conjunction hold. Some SMT solvers use a SAT solver, by first issuing its corresponding SAT problem to the SAT solver, getting a truth assignment to each atom, and then trying to find an assignment to free variables that satisfies every truth assignment.[20]

Thus, an SMT solver is a program which given an SMT problem it will report `sat` `unsat` or `unknown`. The result may be `unknown` due to timing constraints—a definite answer could not be found in time—or due to the problem being undecidability. In practice, the former is more often the case, since often SMT solvers make use only of theories under which the problems are decidable.

SMT solvers are useful in formal verification because they are fast and can be used for a common subset of problems that a more specialized verification system may encounter. The verification system can specialize in the deductive verification specific to the computer program being verified, while facts about arithmetic, strings or byte arrays can be delegated to the SMT solver. Think for example about the situation when, deep in program execution, an `if`-statement is encountered, with the condition `x + y > 0`. `x` and `y` may be unknown values but have many conditions apply to them which have been accumulated during the course of execution. If the SMT solver, when given all the accumulated conditions on the variables x and y and the claim $x + y > 0$, comes back with the result `unsat`, the body of the `if` is skipped. If the negation of the formula, $\neg(x + y > 0)$ comes back `unsat`, the body is executed. If both queries come back as `sat`, then under the given conditions, both execution paths are possible.

An SMT solver may also help to check postconditions: if the postcondition is negated and the SMT solver reports `unsat`, then we know the postcondition always

[20] In principle, the $\mathbb{K}$ prover follows a conceptually similar strategy, where it asks the SMT solver to come up with solutions to a less general problem (intermittent SMT problems) to solve a more general problem (whether a program satisfies a specification).

holds, whereas if it comes back `sat` we know it does not. This project will exclusively make use of the SMT solver Z3[11], which is used in the $\mathbb{K}$ framework's deductive program verifier.

1.7.3.4 Symbolic execution[21]

Symbolic execution is executing a program, only with variables instead of concrete values in some places.

Think of it like this: semantics are a description of how a program should compute its result. Computation and calculation are just two sides of the same coin. Given the program `a = 2; b = 3; c = 5; print(1/ (c * (b + a)))`; we can compute that the program will output 25. No matter what the program looks like, we can always run it and see what it produces, just like we can compute any arithmetic expression, e. g., $(2 + 3) * 5$.

Symbolic execution is to computation what algebra is to calculation. Given the expression $(X + (X + 1)) * (X + 2)$, we can't compute a result. We can only rewrite it, for example to $2X^2 + 5X + 2$. What we can do, however, is to make deductions about the expression. And since the concrete expression, with values 2, 3 and 5, is just a special case of the algebraic expression, anything we can deduce about the algebraic expression is also valid for the concrete expression, and all expressions like it. For example, we know that for any value of $X > -\frac{1}{2}, (X + (X + 1)) * (X + 2)$ is positive. We can also figure out if there are any interesting edge cases, like if the result can be 0, which it can (at -2 and $-\frac{1}{2}$).

In symbolic execution, instead of fully evaluated results, we end up with symbolic results. For example, take a look at the following program, where we assume integers are 32 bit. There are certain "dangers" lurking in it.

Listing 1.2: A small C function to execute symbolically.

```
1  function foo(int a, b, c) {
2    assert (a > b, b >= 0)
3    if (c != 0) {
4        d = c * a
5        if (b == -a) {
6          return d;
7        } else {
8          d = d + b * c;
9          return 1 / d;  // Danger zone.
10       }
11   }
12   return 0;
13 }
```

Let's say we get past the assertion at the start. Executing this program symbolically, we realize it can return 0, $c * a$, or $\frac{1}{c * (b + a)}$, with a, b and c unknown. Now

[21] Parts of this section has been previously published by the author in a blog post, available here:
`https://medium.com/dlabvc/kwasm-a-new-executable-semantics-for-the-blockchain-14e1bca8a360`.

we want to decide if this division is safe. It obviously is not if a, b and c can be any value. But it is rare that variables are completely unconstrained. For example, in this case the assert and if-conditions guarantee that c != 0 and that a > b >= 0, in which case, things look safe. But the programmer might have missed the possibility of overflow: if the integer values are 4 bytes, $c = 2^{31}$, $b = 0$ and $a = 2$, we get a division by 0.

It is often hard to catch these cases with testing. In a toy example like the one above, a verifier can use a pen and paper to make sure they covered every possible result. With larger programs, it is easier (and faster) to use symbolic execution to get algebraic expressions of the possible states a program can have when it terminates.

1.7.4 $\mathbb{K}$ Framework

The $\mathbb{K}$ framework is a software suite that creates several different, common tools from a single language definition. The syntax and semantics of a language are expressed in the formal language of $\mathbb{K}$, in one or more *modules*. For example, it is common that the syntax and semantics of the language are defined in different modules, but not necessary. KWasm mixes syntax and semantics in files, so that the semantics of a language construct are given immediately after its syntax, for readability reasons.

The syntax of a language is given in a modified version of Extended Backus-Naur form[34], importantly supporting annotating productions. The semantics are given as a *configuration*, expressed as a subset of XML[8] with each tag being a different *cells*, and a set of small-step operational *rules*, which only need to mention the parts of the configuration they make use of. The semantics are expressed in a similar way to rewriting logic[24], but the rules express concurrency and can be interleaved[33, Sect. 2.2]. $\mathbb{K}$ even supports *configuration composition*: defining different parts of the semantics in different modules with their own configurations, then inserting the top-level cells of the different semantics as cells in a new top-level cell. Due to the rules only mentioning the parts of the configuration it uses, as long as the composed configurations do not have any cell names in common, the rules can be carried over unambiguously to the composed semantics.

Examples of defining syntax and semantics can be seen in Chap. 1.8. Configuration composition is used to define Ewasm, described in Sect. 2.3.

The $\mathbb{K}$ framework explicitly states as its goal to be a universal framework for programming language definition. [33] Indeed, several mainstream programming languages have been defined in $\mathbb{K}$, including C [13] [18], Java [7], JavaScript [28], and Solidity [23]. Most importantly for the blockchain use case, perhaps, is the $\mathbb{K}$ formalization of the EVM—KEVM was the first full description of the EVM in a formal verification framework.[19]

Fig. 1.1 shows a conceptual overview of the $\mathbb{K}$ framework: from a single definition, many tools can be generated. These are tools that otherwise tend to be implemented on a per-language basis. The vision of the $\mathbb{K}$ framework is that all tools should be created only once, and then parameterized by language, instead of hard-coding the language definition into the tool.

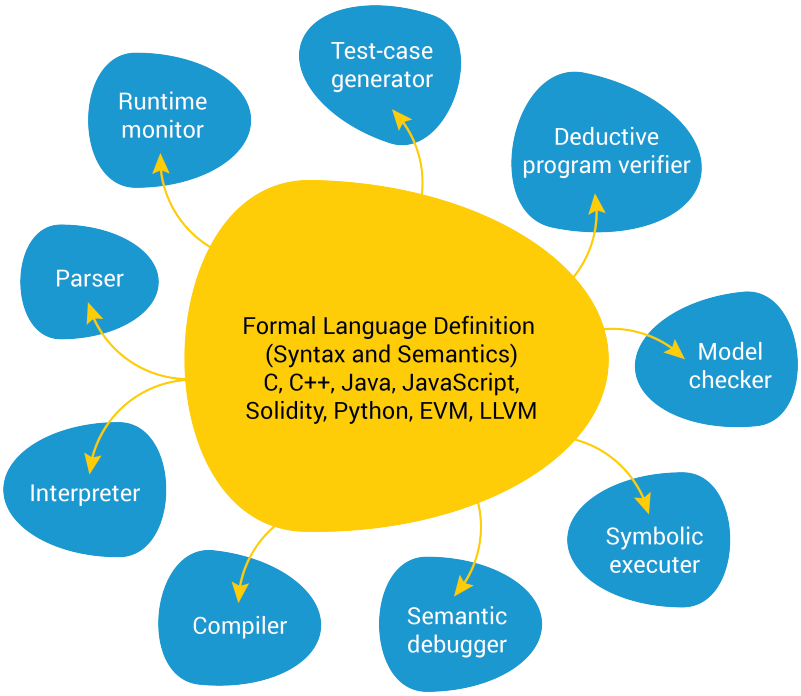


Figure 1.1: Conceptual description of the $\mathbb{K}$ framework. The yellow bubble is the input into the framework, and the blue bubbles are the generated tools. By Runtime Verification, Inc., with permission.

1.7.4.1 The Deductive Program Verifier

From the specification several tools can be automatically generated by the framework. The most important tool for the proposed project is a deductive program verifier, `kprove` `kprove` treats all rewrite rules in the semantics as axioms. The proof obligations are then expressed as separate rewrite rules. Using symbolic execution, applying all possible rewrite rules in turn and branching when several apply, the verifier can automatically prove or disprove that a program, given some pre-conditions, will satisfy some post-conditions. When there are no side conditions the prover uses unification to decide which rules could apply in any given state. However, side conditions may impose extra constraints. To decide whether there is any way to assign concrete values to symbolic variables so that those constrains are met, a query is discharged to Z3, the SMT solver. If it comes back `unsat` the rule can be ignored, and if not, it will get applied. In the end, Z3 is used again if our proof obligation dictates that the variables in the final configuration must satisfy certain constraints. Then, the end constraints are negated, and only if Z3 reports that this is `unsat` is the proof complete.

In theory, the fully mechanized semantics of a language should be enough to prove properties of programs. However, previous experience with KEVM shows that in reality, this is not true for verifying realistic programs due to the the limitations of `kprove` and of Z3.[29] The main limitation consist in $\mathbb{K}$'s inability to reach conclusions about integer arithmetic—this is delegated to the SMT solvers for all expressions which are not trivially equal from a structural perspective—and in Z3's inability to understand the domain objects of $\mathbb{K}$, as well as efficiently reasoning about certain modular arithmetic which is common in KEVM—and, as we shall see, in KWasm. The result is that whenever the prover reaches a decision point where the decision depends on equalities or inequalities in arithmetic, but the formulas also contain

domain objects, the prover is stuck. To cope with this, one can add new axioms to the semantics for the prover to apply that it in principal should be able to deduce on its own—i.e. taking care to not extend the theory (by making new things true) in the process.

The underlying logic of the $\mathbb{K}$ deductive verifier is all-path reachability logic[35], a subset of matching logic[32]. All-path reachability deals with the question of whether on every terminating path of execution in a symbolic and possibly non-deterministic program a certain state will eventually be reached. The program and the end state both take the form of a pattern with side conditions, and thus the resulting proofs show that any program matching the initial pattern will eventually reach a state where it matches the end pattern. The logic is designed to take the transition rules in an operational semantics (rewrite rules) as axioms, while the rules of the logic allows reasoning about non-determinism. Using reachability logic or the more general matching logic is one alternative to Hoare logic (see Sect. 1.7.3.1).

1.7.4.2 The $\mathbb{K}$ Language

In $\mathbb{K}$ syntax productions are declared in the following way.

Listing 1.3: Basic syntax declarations in $\mathbb{K}$.

```
1  syntax Var ::= Id
2  syntax Expr ::= Int | Var
3                | Expr "-" Expr
4                | Expr "+" Expr
5
6  syntax Expr ::= Expr "==" Expr
7
8  syntax IntList ::= List{Int, ":"}
```

This creates the `Expr` and `Var` sorts, and two new terminals, `-` and `+`. We can extend a sort in several different syntax declarations, as is done with `Expr` above.

A special type of production is `List`, which creates a list sort from a sort and a delimiter string. So for example, the `IntList` syntax declarations declares a list of integers delimited with `:` This would tell the parser that the string "1 : 2 : 3" could be a list. Inside $\mathbb{K}$ the list is terminated by a nil value derived from the sort name—the above type of list would have the nil value `.IntList`.

Furthermore, syntax productions can have extra annotations. For example, take the following production:

Listing 1.4: Syntax annotations in $\mathbb{K}$.

```
1  syntax Int ::= sum(IntList) [function, functional, smtlib
      (intSum)]
2  rule sum(.IntList) => 0
3  rule sum(I : IS) => I +Int IS
```

The `function` annotation means that the rules using this production apply anywhere and immediately. The rules can thus not mention cells or any other context, but must all data available to it passed as parameters. The `functional` annotation says that the function is total. It is up to the implementer to ensure this is the case.

The `smtlib` annotation tells the SMT solver to declare this as a function in SMT world, rather than treating an unevaluated `sum(X)` as a separate constant in every instance. The rule for the above declared `sum` function would be: The function's behavior is specified with the `rule` keyword, where each rule may use pattern patching (as we do here) or with side conditions, or both.

A semantics can have one or more[22] configurations, which describe the runtime structure. The structure is an extensible tree with typed nodes, described using an XML-like syntax. Typically, there is a `<k>` cell which can hold values of any sort (they are all subsorted under the `K` sort) where the initial program is put. For example, a simple stack-based language may have this configuration:

Listing 1.5: Example of a $\mathbb{K}$ semantics configuration.

```
1   configuration
2     <k> $PGM:Expr </k>
3     <stack> .IntList </stack>
4     <vars> .Map </vars>
5     <storage>
6       <location multiplicity="*" type="Map">
7         <index> 0 </index>
8         <value> 0 </value>
9       </location>
10      <size> 0 </size>
11    </storage>
12    <result> 0 </result>
```

The `multiplicity` and `type` annotations mean that the `<vars>` cell contains any number of `<var>` cells, and that the first cell in `<var>`, in this case `<name>`, can be used for lookups. Internally this structure is turned into a map with `Identifier`s as keys and `<var>` cells as values. The map is initially empty.

The operational semantics in $\mathbb{K}$ is given with *rules*. Rules have left-hand sides, right-hand sides, `requires`-clauses and `ensures`-clauses. We already saw examples of rules when we defined the `sum` function above. They had with left-hand and right-hand sides separated by `=>` and no condition clauses. The main differences between function rules and operational rules are that the latter may mention cells, and are treated as the steps of the interpreter or virtual machine, that the prover may branch on. For example, we could imagine rules using the configuration and syntax we have already defined may look like this:

Listing 1.6: Operational rules in $\mathbb{K}$.

```
1   rule <k> V:Var => N ... </k>
2        <vars> ... V |-> IDX ... </vars>
3        <location>
4          <index> IDX </index>
5          <value> N </value>
6        </location>
7      requires IDX >Int 0
```

[22] Typically there is only one configuration per semantics. Several configurations are only used in practice with configuration composition.

```
8
9       rule <k> N:Int => . ... </k>
10           <result> _ => N </result>
```

Let us consider the first rule. The `requires`-clause ensures we are only accessing non-zero addresses, meaning this rule does not deal with null pointers in our language. The `...` at the end of the `<k>` cell means the cell contains an associative list with `~>` as the separator and `.` as the identity. `<k> X => Y ... </k>` is syntactic sugar for `<k> X ~> DOTVAR => Y ~> DOTVAR </k>`, or, equivalently, `<k> (X => Y)~> DOTVAR </k>`. In interpreting the rules, the parentheses group to the right, but the list can be appended to from either end. The double `...` around the map indicate that map is an associative-commutative structure of `_ |-> _` key-value pairs, and that we match anywhere in that structure. Equivalently, we could have called matched on the `<vars>` cell with the pattern `<vars> VARS </vars>` and replaced `IDX` with `VARS[N]` in the rest of the rule. Note also that we only mention the cells we are accessing, and that the left-hand side consists of all the left-hand sides of `=>` in the cells which have this arrow, and vice versa for the right-hand sides. The cells which are not mentioned, or which do not contain a `=>` symbol, have identical left-hand sides and right-hand sides, meaning they do not change.

The next rule uses two more features: the `_` wildcard, and the `.` identity of the `~>` associative list. This rule gives our language the feature that the last computed number is the result, and is put int the `result` cell.

The syntax, configuration and rules can be grouped in *modules*. Modules can import other modules, which is essentially like a C import: it is as if the entire contents of the imported module was pasted in place of the import. For example, the `DOMAINS` module contains much of the $\mathbb{K}$ standard library for maps, integers and strings. A common practice[23] is to separate the syntax and the semantics into separate modules, the latter importing the former. For example, the little toy language we have defined in this section could look like this, spread over two modules:

Listing 1.7: An example of a complete $\mathbb{K}$ semantics, split over two modules.

```
1  module K-PRIMER-SYNTAX
2      imports DOMAINS
3
4      syntax Var ::= Id
5      syntax Expr ::= Int | Var
6                    | Expr "-" Expr
7                    | Expr "+" Expr
8
9      syntax Expr ::= Expr "==" Expr
10
11     syntax IntList ::= List{Int, ":"}
12
13 endmodule
14
```

[23] We deviate from this practice in KWasm by importing in the reverse direction, importing the semantics module into the syntax module. This is to avoid parsing issues.

```
15 module K-PRIMER
16     imports K-PRIMER-SYNTAX
17     configuration
18       <k> $PGM:Expr </k>
19       <stack> .IntList </stack>
20       <vars> .Map </vars>
21       <storage>
22         <location multiplicity="*" type="Map">
23           <index> 0 </index>
24           <value> 0 </value>
25         </location>
26         <size> 0 </size>
27       </storage>
28       <result> 0 </result>
29
30     syntax Int ::= sum(IntList) [function, functional, smtlib
          (intSum)]
31     rule sum(.IntList) => 0
32     rule sum(I : IS) => I +Int IS
33
34     rule <k> V:Var => N ... </k>
35          <vars> ... V |-> IDX ... </vars>
36          <location>
37            <index> IDX </index>
38            <value> N </value>
39          </location>
40        requires IDX >Int 0
41
42      rule <k> N:Int => . ... </k>
43           <result> _ => N </result>
44
45      // K uses C-style comments.
46      /* Here is
47      a multiline comment. */
48
49 endmodule
```

1.8 The KWasm Project

At the outset of this project, KWasm supported most of the functionality in a single Wasm module: declaring and using functions, memories, tables and globals, and putting almost all instructions (except certain floating-point conversions) in a function body. In Chap. 2 we describe how we added support for interacting modules, along with some other refactoring and code improvements and more thorough testing. In this section, we give an overview of KWasm, its design, and its relationship with the official specification of Wasm.

1.8.1 Design

The basic design of KWasm follows that of the official specification of Wasm, which uses a very similar formalism to that of $\mathbb{K}$. We have only chosen to deviate from Wasm in one way: instructions can be executed directly, like in regular assembly language, without invoking a function. The KWasm program starts with an empty stack, so all stack-based operations can be executed directly. The rest of the configuration contains minimal default values.

The execution semantics is split up in four main files: `data.md`, `numeric.md`, `wasm.md` and `wasm-text.md`. Each uses a literate programming style, interleaving Markdown documentation and $\mathbb{K}$ code. `data.md` holds modules that define data structures that are needed for the semantics, but are in principle decoupled from Wasm, such as stacks and byte maps. `numerics.md` contains numerical functions described in [31, Sect. 4.3] that instructions lift their operands through, such as `eq`, `add` and `xor`. `wasm.md` contains the bulk of the abstract syntax and semantics as described in [31, Sect. 4.4, 4.5]. `wasm-text.md` contains text-only syntactic sugars as described in [31, Chap. 6]. In the situations where the text format syntax and abstract syntax overlap or are one-to-one, they are put in `wasm.md` rather than in `wasm-text.md`.

1.8.2 Wasm and KWasm Introduction

Wasm is a stack-based language with high-level features such as functions and memories with bounds checks. All Wasm code is structured as modules, which is a set of structures that have access to each other—functions that can call each other, memories that can be accessed, etc. Some of these may be declared in the module itself, while others may be imported. KWasm executes a module by sorting its list of declarations (described in Sect. 2.1) and putting them in the `<k>` cell. The only place where regular, assembly language-like instructions appear in Wasm is in function bodies. When a function is invoked, its body is loaded at the top of the `<k>` cell.

So the `<k>` cell holds lists of instructions and lists of declarations. We order instructions and declarations into the supersort *statements*, and list of instructions of declarations into lists of statements[24]. The lists are consumed by taking the first element of each and sequencing them with the rest of the list using the associative list operator `~>`. This process is called "heating" in $\mathbb{K}$. Then rules can match on the head of the `<k>` cell, with `...` occluding the rest of the list.

Listing 1.8: The rules for sequencing statements in KWasm

```
1     syntax Stmts  ::= Instrs | Defns
2   // --------------------------------
3     rule          <k> .Stmts          => .        ... </k>
4     rule          <k> (S:Stmt .Stmts) => S        ... </k>
```

[24] All the lists—of declarations, of instructions, and of statements—are produced by the parser and then consumed, never extended. Due to the operation of taking from a list being covariant in the list sort, we can treat a list of declarations as a list of statements.

```
5     rule [step] : <k> (S:Stmt SS)      => S ~> SS ... </k>
         requires SS =/=K .Stmts
```

1.8.2.1 Instructions

Instructions manipulate the stack by pushing constant values onto it, or by pushing call frames or branching targets (called "labels"). While the Wasm specification keeps everything on a single stack and regular Wasm implementations use different stacks for each, KWasm uses two stacks: one for constants, and one for everything else. The stack for numeric constants is called `<valstack>` in KWasm. A `Val` has the form `< TYPE > VALUE` in KWasm, the `TYPE` being `i32` or `i64` for integers, or the special case `undefined`. Numeric instructions return a `Val`, where an `undefined` may result from an operation such as division by 0.

Listing 1.9: Pushing constants in KWasm.

```
1     syntax PlainInstr ::= IValType "." "const" Int
2  // ----------------------------------------------
3     rule <k> ITYPE:IValType . const VAL => #chop (< ITYPE >
         VAL) ... </k>
4
5     rule <k> undefined => trap ... </k>
6     rule <k>         V:Val    => .          ... </k>
7          <valstack> VALSTACK => V : VALSTACK </valstack>
8       requires V =/=K undefined
```

Constants get pushed to the stack by being lifted through the `#chop` function, which returns a `Val`. This allows negative integers or integers outside the range of the type to be given to the `const` instruction and turned into their representative in the range from 0 to $2^{|\text{ITYPE}|-1}$ The resulting `Val` is pushed to the `<valstack>`.

An example of consuming the stack is binary operations, such as `i32.add` or `i64.eq`. These can be abstracted in the following way.

Listing 1.10: Popping constants for binary operations in KWasm.

```
1     syntax PlainInstr ::= IValType "." IBinOp
2  // ----------------------------------------
3     rule <k> ITYPE . BOP:IBinOp => ITYPE . BOP C1 C2 ... </k>
4          <valstack> < ITYPE > C2 : < ITYPE > C1 : VALSTACK =>
               VALSTACK </valstack>
```

This rule consumes two operands and passes them to the function `.` which takes a type, a binary operation-representing atom (such as `add`) and the operands, and results in a new `Val`.[25]

All functions that return a `Val` respect the custom that `Val`s should be in the correct range for their type. Therefore, the only place where integers in the wrong range can appear is in the `const` operations. Throughout the semantics, it is assumed

[25] This is indeed a function, declared with the production `syntax Val ::= IValType "." IBinOp Int Int [klabel(intBinOp), function]`.

that `Val`s respect this invariant. This assumption will be made explicit when we cover proving in Chap. 3 with the predicate `#inUnsignedRange(TYPE, NUM)`.

In the text format of Wasm instructions can be *folded*, allowing operations to be written in an S-expression-like form by wrapping them in parentheses and giving them 0 or more "arguments". Non-folded instructions are called *plain* instructions, as seen in listings 1.9 and 1.10 in the sort name `PlainInstr`. So for example, the following sequences of instructions are desugared to the same sequence:

- `i32.const 1 i32.const 2 i32.add`
- `(i32.add (i32.const 1)(i32.const 2))`
- `i32.const 1 (i32.add (i32.const 2))`
- `i32.const 1 i32.const 2 (i32.add)`

Listing 1.11: Rules for unfolding folded Wasm instructions.

```
1     syntax FoldedInstr ::= "(" PlainInstr Instrs ")"
2                          | "(" PlainInstr        ")" [prefer]
3   // ----------------------------------------------------------
4     rule <k> ( PI:PlainInstr IS:Instrs ):FoldedInstr => IS ~>
          PI ... </k>
5     rule <k> ( PI:PlainInstr           ):FoldedInstr =>
               PI ... </k>
```

The `[prefer]` annotation is an instruction to the parser.

1.8.2.2 Declarations

Here is an example of a function declaration in Wasm:

Listing 1.12: A Wasm function declaration.

```
1 (func $main (result i64) (i32.const 42) (i64.const 10) (call
     $set-and-return))
```

The `func` keyword, the (optional) identifier `$main` and the type declaration `(result i64)` make up what may be called the function header in other languages, while the rest of the declaration make up what might be called the function body. The body is made up of a list of instructions.

Central to Wasm is the concept of the *store*. This is where all functions, memories, tables, and globals live. The module is simply a collection of pointers into the store. If the store is S and the list of functions is S.`funcs`, which has length n, then when we declare the function in listing 1.12, the list S.`funcs` gets extended, and pointer is added to the module that indicates that the `$main` function can be found at location n in the store. Inside a module, a function (or other allocated element) is referenced by its *index* . If the function in listing 1.12 is the first function to appear in the module, it gets index 0. The next function will get index 1, and so on. Each index then points to an *address* in the store.

1.8.2.3 Runtime Structure

The complete configuration of the KWasm is given in App. C. Below we give an abridged version, focusing on the parts necessary for dealing with functions, with

`<...>` in place of omitted cells.

Listing 1.13: Abbreviated KWasm configuration.

```
configuration
      <wasm>
        <k> $PGM:Stmts </k>
        <valstack> .ValStack </valstack>
        <curFrame>
          <locals>    .Map </locals>
          <curModIdx> .Int </curModIdx>
          <...>
        </curFrame>
        <moduleInstances>
          <moduleInst multiplicity="*" type="Map">
            <modIdx>      0    </modIdx>
            <types>       .Map </types>
            <funcIds>     .Map </funcIds>
            <funcAddrs>   .Map </funcAddrs>
            <...>
          </moduleInst>
        </moduleInstances>
        <mainStore>
          <funcs>
            <funcDef multiplicity="*" type="Map">
              <fAddr>    0              </fAddr>
              <fCode>    .Instrs:Instrs </fCode>
              <fType>    .Type          </fType>
              <fLocal>   .Type          </fLocal>
              <fModInst> 0              </fModInst>
            </funcDef>
          </funcs>
          <nextFuncAddr> 0 </nextFuncAddr>
          <tabs>    <...> </tabs>
          <mems>    <...> </mems>
          <globals> <...> </global>
        </mainStore>
        <...>
      </wasm>
```

The `<curFrame>` cell informs us about in which module we are currently executing (may change when we call an imported function) and what local variables are available. The `<curModIdx>` cell is used to look up the correct module in the set of modules. A module instance has mappings between indices and addresses, as well as some extra mappings between identifiers and information about the different types of the functions in a module. The global store contains the actual functions, where each function has an address by which we reference it (growing from 0, where each new allocated function gets the next available address), a body, a type, a definition of which local values it has available, and a reference to which module it belongs to. The module index is necessary to know, for example, which memory should be

modified when invoking the `i32.store` operation.

2

Part 1: Completing and Extending KWasm

To use KWasm for verifying things like host function calls and smart contracts, we need to complete KWasm and extend it with an embedding. Part 1 of the project is to make these completions. We start by adding support for modules, imports, and exports. In the process, we also found that with full module support we could test our implementation against official conformance tests with little extra effort, which helped locate some missing functionality and revealed some issues with the test suite that makes it non-ideal for our purposes. Finally, we add a prototype EWasm embedding by composing an Ethereum client semantics with KWasm and connecting them through a thin synchronizing interface.

2.1 Giving KWasm Support for Modules

KWasm was developed from the bottom up, starting with basic stack operations and adding higher-level features such as function calls, tables, and memory. Yet Wasm programs always come packed in modules, and interaction between several modules is a core Wasm functionality. The final major step in completing KWasm has been to allow support for having several, full modules.

A Wasm module may contain any of the following:

- functions
- named types
- a memory
- a table
- globals
- table initializations
- memory initializations
- imports
- exports
- a start function.

The simplest possible module is simply `(module)`, which contains none of the above, while a module containing all of them may look like follows:

Listing 2.1: A simple Wasm module

```
1 (module
2   (global $x (import "otherModule" "g1") i32)
3   (func $init
```

```
4       (i32.store (i32.const 0) (global.get $x))
5     )
6     (type $main-type (func (result i64)))
7     (func $main (export "main") (type $main-type)
8       (i64.load (i32.const 0))
9     )
10    (memory 1)
11    (table $tab funcref (elem  $init $main))
12    (export "myTab" (table $tab))
13    (start $init)
14  )
```

2.1.1 Finding the Correct Order of Definitions

Each memory, table, etc., in KWasm is *defined*, or *declared*,[1] and is then available for use. However, the text format of Wasm allows declarations to come in any order inside a module. However, some declaration types depend on others, so the first step to instantiating a module is ordering the declarations. The official specification's description of module allocation and instantiation[31, Sect. 4.5] proceeds sequentially from the structured module format, where all declarations are available. KWasm consumes modules based on the text format, applying syntactic sugars as they are encountered, in an interpreter-like fashion. Some declarations can be made inline. For example, a function export can be specified in the function header, or an import can be declared as a regular function without a body and with an import statement.

To the best of our knowledge, KWasm is the only Wasm implementation to interpret the Wasm text format, where most other representations either compile the code, interpret the binary format, or first convert concrete syntax into the abstract syntax before interpreting. In KWasm, we try to use the text format as given, desugaring and reordering in a streaming fashion as much as possible. Therefore, in KWasm declarations are handled as they are encountered[2]. However, declarations may depend on each other. Some declarations are *allocations* and create a new structure—table, function, memory, or global—in the global store, and a pointer in the current module to that location in the store. Other declarations are *initializations*, which modify the contents of structures in the global store. Then there are export and import declarations, which create names for store pointers, and import those pointers by name.

[1] In Wasm, function, memory, table, global, import and export declarations are called "definitions", and the naming of the sorts in the KWasm specification will reflect this. However, we find that the term "declaration" is a clearer description, and will prefer to use it.

[2] Note that "interpreting the text format" is not perfectly sound. The top-level `(module ...)` may be omitted in a text format source file. However, in KWasm, each declaration is processed as it is encountered, and declarations only get sorted when wrapped in a top-level `(module ...)` constructor. So, for example, the valid source text `(export "foo" (func 0))(func)`, where the export precedes the function definition, cannot be processed by KWasm. A future improvement proposal is to perform a first pass over the parsed source to desugar the text format into the core AST format. One transformation of this first pass would be to ensure there is a top-level `(module ...)` to each module. Another possible is approach is to push declarations that cannot be processed right now to a "waiting" stack, to be processed before finishing the module.

These declarations have certain natural dependencies on each other. To determine in which order declarations need to be handled we studied the official specification and cataloged the dependencies. Fig. 2.1 shows the dependency directed acyclic graph between different types of declarations. At the bottom of the hierarchy are the allocations—globals, functions, tables, and memories—and imports, which are always declared with their full specification present. Exports rely on these allocations since what they export must have been allocated (and thus given an index within the module, and an address in the store). However, exports do not require any initialization, since they only need to know what index they are exporting, not what is at the address the index points to. All initializations, unsurprisingly, depend on what they initialize having been allocated. Thus, `data` depends on memory having been allocated (or imported), `elem` depends on a table having been allocated (or imported) as well as the function it is inserting, and initializing globals require that they have been allocated (or imported). At the bottom of the hierarchy of initializations is initializing globals. The reason is that both `elem` and `data` initialization may make use of *constant expressions*, which is any sequence of instructions where each element is either of the type `t.const` *or* of the type `global.get`, where the global value being accessed has mutability `const`, i.e. it is immutable.[31, Sect. 3.7][3] Finally, initializing a module with a start function means running that function, the body of which can contain arbitrary code that may access memory, tables, and globals and call other functions. Therefore, the start function may only be run when the rest of the module is fully instantiated.

In handling a standard Wasm program, we encounter top-level modules, wrapped in `(module ...)`, containing declarations in any order, which we need to make sure we handle in the right order. Furthermore, the text format allows nesting declarations in a sugared way, for example allowing declaring the export of a function in the definition of that function. Therefore we need to make sure that

1. when instantiating a module, its declarations are grouped, and
2. when desugaring nested declarations, it is turned into a sequence of declarations,

so that each declaration has its dependencies fulfilled before being handled.

2.1.2 Grouping the Declarations in Top-Level Modules

When encountering a top-level module, we group the declarations in a module according to sort. The first step is to make a different sort for each type of declaration, and give them a common super-sort. For example, the syntax declarations for function declarations have the following form.

```
1    syntax Defn     ::= FuncDefn
2    syntax FuncSpec ::= TypeUse LocalDecls Instrs
3    syntax FuncDefn ::= "(" "func" OptionalId FuncSpec ")"
```

[3] Currently, a constant expression may only access imported globals, a constraint that may be lifted in the future. We do not enforce this constraint in KWasm. This may lead to trouble in an invalid module, since a global initialization expression may contain `global.get`s of uninitialized globals. According to the basic principles of KWasm, this is not a problem, since we assume we are only dealing with valid modules, and thus only imported globals can be used, which must necessarily have been initialized.

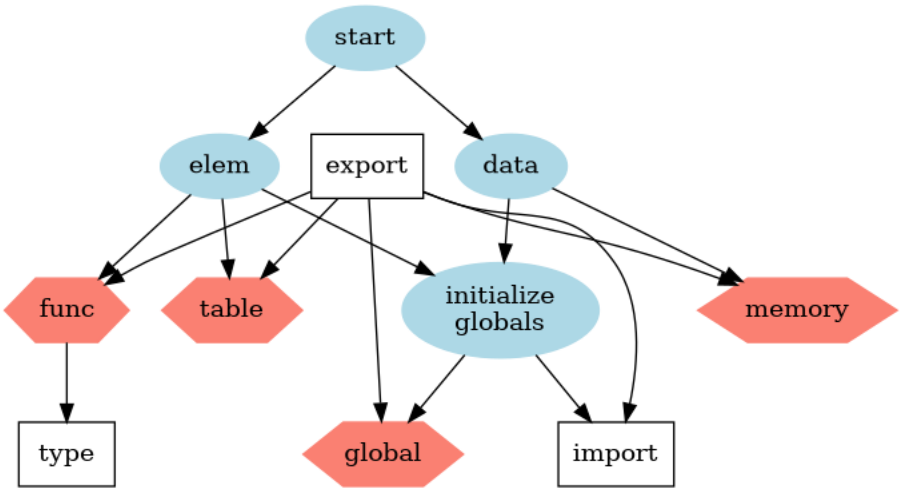


Figure 2.1: This directed graph shows how the different parts of a module's declarations depend on each other. Red hexagonal nodes indicate *allocations*, which create a new structure in the store; blue elliptical nodes indicate *initialization*, which alter the contents of that structure; and white boxed nodes indicate definitions that only alters mappings in the current module. For example, an export depends on the field it exports having been allocated (so that its index is known).

A module is defined to be a series of declarations. These declarations are first grouped by type through an auxiliary function and turned into a structured format, and then the declarations are carried out in an order that respects the above dependency graph.

```
1     syntax Stmt       ::= ModuleDecl
2     syntax ModuleDecl ::= "(" "module" OptionalId Defns ")"
3                         |   "module" OptionalId Map
4   // ------------------------------------------------
5     rule <k> ( module OID:OptionalId DEFNS ) => sortModule(
          DEFNS, OID) ... </k>
6
7     rule <k> sortedModule(... id: OID, types: TS, importDefns
          : IS, funcsGlobals: FGS, allocs: AS, exports: ES,
          inits: INIS, start: S)
8            => TS ~> IS ~> FGS ~> AS ~> ES ~> INIS ~> S
9          ...
10         </k>
11         <curModIdx> _ => NEXT </curModIdx>
12         <nextModuleIdx> NEXT => NEXT +Int 1 </nextModuleIdx>
13         <moduleIds> IDS => #saveId(IDS, OID, NEXT) </
              moduleIds>
14         <moduleInstances>
15           ( .Bag
16          => <moduleInst>
17               <modIdx> NEXT </modIdx>
```

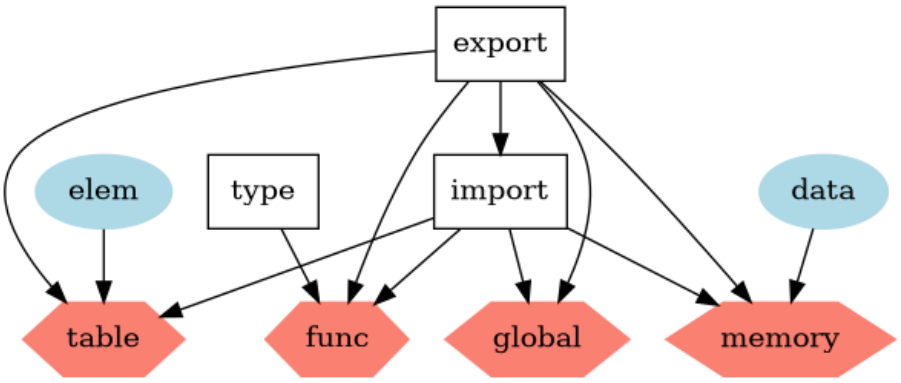


Figure 2.2: This directed graph shows which declarations can be inlined in another. For example, a table or memory can have their initializing declarations inlined, and any allocation can be exported inline.

```
18                  ...
19                </moduleInst>
20              )
21              ...
22            </moduleInstances>
```

The `sortedModule` rule uses record syntax: the sorting function returns a record, which can be accessed with a leading `...` symbol and field names followed by a `:`, such as `types: TS`.

It is important that the grouping respects the internal order of each type of definition since their order is used to assign them indices that they can be referred to by internally.

2.1.3 Desugaring Inline Declarations

Due to the above-mentioned sugaring, the module sorting will not put all `exports`, `import`, `elem` and `data` declarations in the correct order. For example, any allocation may have an inlined import and any number of inlined exports, and memory and table declarations may have inlined `data` or `elem` declarations, respectively. [31, Sect. 6.6].

Fig. 2.2 shows which declarations can be inlined inside another. If we want to be able to desugar these declarations as we encounter them, we need to make sure that all the dependencies of the inlined declaration are in place at the time it is encountered. We can check this in a structured manner by looking at each edge in Fig. 2.2 and convincing ourselves that it can be suitably dealt with.

Every edge originating in the `export` node can be dealt with by performing the allocation or import first, obtaining an index and exporting it. Due to the initial grouping of the different declarations in a module, `imports` cannot be dealt with as they are encountered, because the order of imports must be respected, and desugaring after grouping different kinds of declarations could mean that, for example, a `(func ... (import ...))` would always get reordered so that it would appear after an `(import ... (func))`. Therefore we make use of the `[macro]` annotation of

𝕂 and immediately desugar these imports.[4]

The arrows from the `elem` and `data` nodes are dealt with by sequencing definitions so that the allocation comes before the initialization. Here, too, we must ensure that the order of `data` and `elem` segments is not changed since they should be processed in given order in the module. For example, the following two Wasm modules are equivalent and represent a sugared, desugared, reordered, and simplified module, respectively.

Listing 2.2: Three equivalent text format modules.

```
1 (module (data (i32.const 0) "ab") (memory (data "cd")))
2 (module (data (i32.const 0) "ab") (data $id (i32.const 0) "cd
     ") (memory $id 1 1))
3 (module (memory $id 1 1) (data (i32.const 0) "ab") (data $id
     (i32.const 0) "cd"))
4 (module (memory $id 1 1) (data $id (i32.const 0) "cd"))
```

In this instance, the first `data` declaration would be overwritten by the second. However, using reordering before desugaring would cause the following modules to be equivalent.

Listing 2.3: Three equivalent modules when (incorrectly) reordering before desugaring.

```
1 (module (data (i32.const 0) "ab") (memory (data "cd")))
2 (module (memory (data "cd")) (data (i32.const 0) "ab"))
3 (module (memory $id 1 1) (data $id (i32.const 0) "cd") (data
     (i32.const 0) "ab"))
4 (module (memory $id 1 1) (data (i32.const 0) "ab"))
```

Note that the starting from the same module, if reordering comes before desugaring, the memory contains the byte representation of `"ab"` in the end, rather than `"cd"`.

The official specification instructs us to do this by placing the `data` or `elem` segment right after the `memory` or `table` allocation, introducing an identifier for the memory or table if necessary. Since there can be at most one memory and one table per module, we can choose a module-unique identifier and then perform the expansion, again using the `[macro]` annotation.

The arrow from the `type` node is slightly more complex. In short, types can be declared separately, and functions can then refer to the type by index or name. The below module has functions of two types. The first type is declared explicitly at the module level, while the second type is the implicit. Implicit types are only allowed in the text format.

Listing 2.4: Examples of function type uses.

```
1 (module
2   (type (func (param i32) (result i64)))
3   (func $uses-type-0-explicit (type 0)                    (i64.
       extend_i32_u (local.get 0)))
```

[4] For historical reasons, KWasm has not used `[macro]`, but desugaring text-format declarations would be a good use of them. See this issue for discussion.

```
4   (func $uses-type-0-implicit (param i32) (result i64) (i64.
       const 0))
5
6   (func $declares-new-type    (param i32) (result i32) (i32.
       add (local.get 0) (local.get 0)))
7   (func $uses-type-1-implicit (param i32) (result i32) (i32.
       add (local.get 0) (local.get 0)))
8   (func $uses-type-1-explicit (type 1)                 (i32.
       add (local.get 0) (local.get 0)))
9 )
```

The first two functions both use the same type, which was declared explicitly. One of these references the declared type directly, while the other does so implicitly, by having the same type. The next three functions all have the same type. When a function is encountered in the text format, and its type is not already present as a module-level type, a new module-level type is declared. This is the case for the first function in the second group. The next function reuses the type implicitly, by having the same type signature, while the third reuses this type explicitly: it will have index 1 since it is the second type to be declared in the module.

Dealing with types was handled outside the scope of this project, by other KWasm contributors before this project started. It involves desugaring, checking for the existence of types and reusing them properly, using the correct index or identifier.

With that, all the arrows have been addressed and thus all the ways of inlining declarations in each other can be handled correctly.

2.2 Testing KWasm Against the WebAssembly Core Test Suite

WebAssembly has a reference test-suite for the core semantics, as well as equivalent tests for a JavaScript embedding[5]. These test conformance for an implementation of the core (or unembedded) Wasm semantics. After implementing modules in KWasm it became possible to run these tests to identify missing functionality in KWasm since the tests often declare one or more modules and test interaction between modules. This presented us with some low-hanging fruit, where we could sanity-check KWasm against the core test suite.

This section describes the process of setting up the test harness, the result of running these core tests and what missing functionality was identified. In Chap. 4 we will discuss why we believe the core test suite is unsuitable as a standard test of a pure Wasm implementation.

[5] `https://github.com/WebAssembly/spec/tree/master/test`

2.2.1 The Reference Interpreter

The test suite contains definitions that are not in the official Wasm specification.[6] To some extent this is necessary. Wasm can not be executed on its own—it must be embedded. Thus, the reference interpreter provides an embedding. The embedding has sparse documentation[7], but its core functionality is simple:

- Files are processed from the top down, like a script.
- A module definition both defines a module, validates it and creates an instance of it.
- Modules can be registered with the `( register <string> <name>? )` function. If `<name>` is omitted, the last instantiated module is registered.
- An import `( import <modulename> <name> )` is resolved by looking up the module name in the registry, and importing from the exports in that module.
- Functions can be invoked with a special `( invoke <name>? <exportname> <arguments>* )` and exported globals can be accessed by `( get <name>? <exportname> )`.
- There is a set of assertions which check that calls return specific values (or fail in some fashion), that they trap or that a module definition is somehow bad, either malformed, invalid, or "unlinkable", meaning imports cannot be resolved.
- Modules can be specified as byte arrays with `( module binary <bytestring>* )` or as strings with `( module quote <string>* )`. Since the test files are parsed, malformed modules would break parsing. These special module kinds allow putting off parsing until run-time so that exceptions can be checked and caught using assertions.

The reference interpreter also has two undocumented host modules, called `"test"` and `"spectest"`. Any calls to these can not be represented purely as Wasm code, so an embedding is necessary. We have not yet reverse-engineered these host modules, and therefore the tests invoking the host functions will not pass.

2.2.1.1 The `WASM-TEST` module

We extend KWasm with a module, `WASM-TEST`, that contains commands for writing ad-hoc unit tests, as well as commands from the reference interpreter. The module extends the `Stmt` sort with the `Auxil` sort which includes assertions and methods for invoking function from outside a module.

However, we do not implement all the functionality of the reference interpreter. For example, we do not implement assertions check for invalid or malformed modules, which we do not consider. Other assertions the reference interpreter supports that we simply skip over (rewrite to `.` immediately) are those that check what kind of `NaN` certain functions return; `assert_unlinkable`, which check that imports resolve correctly; and `assert_exhaustion` which ensures that a non-terminating recursive function will eventually cause the runtime to run out of resources since the semantics do not allow tail-call optimization. The arguments for not implementing the

[6] There is, however, an appendix in the specification on a suitable interface between Wasm and an embedding[31, Sect. 7.1]. The reference interpreter implements many of these interface functions.

[7] `https://github.com/WebAssembly/spec/tree/master/interpreter`

`NaN` assertions and `assert_unlinkable` are, respectively, that we have no ambition to have official floating-point number support, and that we assume we are working only with valid (linkable) modules. The reason that we skip exhaustion checking is that it would complicate the semantics with no real benefit: we could add a cell to the configuration with an integer that increments at each function call, and throw a special exception when a certain number is reached; but we do not do any tail call optimization, so resources will eventually exhaust, with failure of the runtime as a result, rather than a catchable exception. Our chosen behavior matches the semantics of Wasm—an infinite recursion without tail-call optimization must eventually exhaust the machine—but the reference interpreter uses a special exception to catch this type of error and test for it.

Perhaps the biggest piece missing from the `WASM-TEST` module is the host modules, `"test"` and `"spectest"`. KWasm, therefore, does not pass the conformance tests which make use of the host modules. 9 of the total 73 test files contain these imports. We have not investigated what would be necessary to add support for such modules, but in private communication with other Wasm implementers, we have been told that they have dealt with this by reverse-engineering the behavior of the reference interpreter.

2.2.2 Discovering Missing Functionality

After implementing an embedding mimicking the reference interpreter, we set up a simple harness. The core test files are written as scripts. If reference interpreter halts at the end of the script with no result, all tests passed. The same is true of the generated KWasm interpreter.

We pass each file in the core test suite gets to KWasm. If KWasm fails to parse the program, the name of the file is appended to file called "unparseable.txt". If the test parses, the test is executed on both the concrete backends, OCaml and LLVM. If the execution does not complete, it is appended to another file, either "unsupported-ocaml.txt" or "unsupported-llvm.txt". The tests that pass are thus added to neither file, and we can inspect the lists of failing tests at our convenience.

Based on these lists we looked for unsupported functionality in KWasm.

2.2.2.1 Missing Floating-Point Representation

The core tests use the Wasm text format. The text format supports declaring floating-point numerals in hexadecimal format, e.g. `(f64.const -0x1p-1)`. $\mathbb{K}$ does not have built-in support for parsing floating-point numbers in this format. We introduced a simple Python preprocessor that turns hexadecimal floats into floats given in decimal notation, and the preprocessor is now part of the KWasm runner.

2.2.2.2 Specifying Modules

By running the core tests that we found instances where incorrect reordering and desugaring were causing incorrect behavior, the likes of which we described in Sect. 2.1.3.

Furthermore, to run the tests we needed to add support for two special ways to declare modules, `(module quote <String>)` and `(module binary <String>)`. These are features of the reference interpreter, not of Wasm. These allow passing a module in one of the two concrete Wasm formats—text and binary—which will result in either the module being decoded and validated and instantiated as usual or an exception being raised if the module is malformed. This is a way to check, inside the reference interpreter, if the parser works correctly. Since KWasm only deals with the execution of valid modules, and the roles of these constructs in the core test suite is to check if a concrete representation is malformed, we do not consider such modules, but simply skip over them.

2.2.2.3 `NaN`s With Payloads

A regular `NaN` value has any sign, the largest possible exponent, and a non-zero significand (a zero-significand makes the value infinity, positive or negative) [9, Sect. 6.2.1]. The *canonical* `NaN` has a significand which has the most significant bit (bit 22 for `f32`, and bit 52 for `f64`) set to 1, and the remaining bits in the significand set to 0[31, Sect. 2.2.3]. However, any non-zero significand is valid. The text format allows specifying a `NaN` with any non-zero payload given as a hexadecimal number. For example, the canonical `f32` `NaN` value can be specified as `(-nan:0x200000)` [31, Sect. 6.3.2].

The floating points of $\mathbb{K}$ do not have support for `NaN` payloads. Converting all `NaN`s to canonical `NaN`s is also not an option: it would preserve the behavior of floating-point arithmetic, but cause incorrect conversions and memory manipulations. While the official specification allows an implementation to always give canonical `NaN`s as the result for most[8] arithmetic operations on `NaN`s[31, Sect. 4.3.3, NaN Propagation], other operations make use of the exact bit pattern of a floating-point number. For example, converting a floating-point to an integer is simply reinterpreting the bits, so that `NaN`s with different payloads will result in different integers. Also, storing floating-point numbers in memory similarly must respect the actual binary representation.

Since KWasm does not yet, and may not ever, have full support for floating-point conversion to integers and to byte representations, the only ways to get an observable difference between `NaN`s with payloads and canonical `NaN`s is through unsupported operations. We can therefore safely, for now, use only canonical `NaN`s by stripping the payload during preprocessing. Any tests that would fail due to this change causing observable changes will fail anyway due to unimplemented behavior.

2.2.3 Conclusions

Setting up the test harness allowed us to address some shortcomings, but did not give any conclusive answers as to whether KWasm is as conforming to the official Wasm specification as we would like it to be. Our choice to not support floating-point numbers causes many of the failures, as does our current lack of implementation

[8] Some arithmetic operations must respect the payload and only change the sign of the input: `fneg`, `fabs` and `fcopysign`.

of the host modules. Until KWasm becomes a complete Wasm interpreter, which may require adding support for the byte format and floating points, we will not be able to pass the remaining tests. We would be helped by more granular test files, where modules are not grouped only by what operations or declarations they concern themselves with, but where each file is also as self-contained as possible, with modules (and their tests) that do not depend on any other modules being given in separate files. The test suite could also be run with a JavaScript embedder, which may require that we first implement support for the byte code representation of Wasm.

2.3 EWasm Embedding

The `WASM-TEST` module was a bare-bones embedding, adding minimal functionality to core Wasm. For a prototype of a more complex embedding for the blockchain use case, we chose to formalize a subset of Ewasm based in the informal description given here. Since the spec is not final, we chose to only implement the functionality that is necessary to execute the WRC20 contract. Our embedding uses *configuration composition* of the Wasm semantics and a prototype Ethereum Execution Interface (EEI) semantics.

Both the subset of Wasm allowed in Ewasm and the EEI are deterministic.[9] That is, for every state there is at most one applicable transition rule. A design goal of the embedding, which we will call "KEwasm", is that the composed semantics, too, should be deterministic.[10]

Determinism is achieved through a simple synchronization mechanism: any time the Wasm code calls out to an EEI host function, they go into `#waiting` mode. The EEI returns control to the Wasm semantics by leaving a `#result` as the only element in the `<eeiK>` cell. For those host functions that return control, there is a rule that consumes the result and takes the Wasm execution out of `#waiting`, and computation in Wasm can proceed.

2.3.1 The Ethereum Environment Interface (EEI)

A smart contract on Ethereum needs to access certain aspects of the states on the blockchain. This is true of any smart contract language. While the KEVM integrated the blockchain-related operations directly into the semantics, Ethereum 2.0 may introduce the concept of *execution engines*, which would imply there could be any number of different smart contract languages available in the future. To accommodate future semantics development we separate the EEI from EWasm so that it may be reused. Some EEI operations are queries, for example:

- querying which address initiated the transaction

[9] If any smart contract had non-deterministic behavior, it would cause a consensus bug on the blockchain, where the state of the virtual machine could not be agreed upon by honest nodes.

[10] In theory, we could just as well allow arbitrary interleaving of certain operations in the two composed semantics, and use synchronization points. But even if we proved confluence of any two possible interleavings of rules the prover would still try all possible interleavings, slowing down the proof process, so we opt to be more strict.

- what address is making the current call (could be another contract or the user which initiated the transaction)
- account balances
- timestamp and height of the current block.

Other operations modify blockchain state, like:

- sending a message
- modify the contract's storage (different from the Wasm memory in the contract)
- emitting messages.

For our purposes, the important aspects of the blockchain state are accessed through the following imported functions:

Listing 2.5: The currently implemented EEI API

```
1 (func $revert (import "ethereum" "revert") (param i32 i32))
2 (func $finish (import "ethereum" "finish") (param i32 i32))
3 (func $storageLoad     (import "ethereum" "storageLoad") (
     param i32 i32))
4 (func $storageStore    (import "ethereum" "storageStore") (
     param i32 i32))
5 (func $getCaller       (import "ethereum" "getCaller") (param
      i32))
6 (func $callDataCopy    (import "ethereum" "callDataCopy") (
     param i32 i32 i32))
7 (func $getCallDataSize (import "ethereum" "getCallDataSize")
     (result i32))
```

All of these, except `$getCallDataSize`, either reads from or modifies the *Wasm* memory. The `i32` parameters specify offset and length of the portion of memory results should be written to or read from. In essence, this makes the Wasm memory play the role the stack and registers play in the calling conventions used in mainstream assembly languages.

`$revert` and `$finish` do not return control to the Wasm execution. Reverting in Ethereum means undoing all state changes made by the transaction, but consuming the gas. Finishing in Ethereum means committing the changes the transaction has made.

The instructions `$storageLoad` and `$storageStore` operates on the persistent storage that each contract has. This is a key-value store mapping, where both keys and values are 32 bytes (256 bits)—essentially a word-addressable 256-bit memory. This convention comes from EVM, which does not have a built-in concept of linear memory, like the Wasm memory. The Wasm memory store is temporary in Ewasm and is reset (zeroed out) after each transaction. Therefore, these functions are needed to access the persistent storage, for example, to store the token balance of an address in the WRC20 contract. This is needed because persistent memory is expensive on the blockchain, as it needs to be replicated by all nodes, and the Wasm semantics dictates that Wasm memory comes in pages of 64 KiB.[11]

[11] At a cost of 20k gas per 256 bit word[40, Appendix G], making a single page persistent would cost $\frac{65{,}536*20{,}000}{32} \approx 41,000,000$ gas. The gas price in ether is set by the transaction sender, but

In the EVM, when a contract is called it starts executing the EVM byte code starting at the first instruction. The *call-data* is a sequence of bytes passed along in the transaction, accessible through special opcodes. A common pattern in smart contract interactions is to call a specific function in a contract by sending a selector in the call-data. The contract is designed so that the first thing it does is check the selector bytes. Each public function in the contract has a specific selector assigned to it, and if the selector bytes in the call-data match any of these, the corresponding function is called. This simulates a contract that can perform several different functions and does so in a way that simulates the different methods of an object in a higher-level language. The selector is the first four bytes of the hash of the function signature, e.g. the Keccak-256 hash of `"balance(address):(uint64)"`[12]. In addition to the selector, the call-data contains the parameters expected by the function. The selector and parameter order is merely a calling convention, called the "Contract Application Binary Interface" (Contract ABI)[1]. However, any contract can in principle use the data any way it sees fit.

In Wasm, there is no single entry point to a module, the way there is for regular bytecode, where the instruction pointer is simply set to the first instruction. Instead, one of the ways that Ewasm constrains what a valid module is[13], is by enforcing that the module exports a function called `"main"`, and that this function takes no parameters and returns no values. Instead, all parameters are passed through call-data and are returned through the sequence of bytes called return-data.

The EEI configuration contains all available data of a transaction, the storage, code, and addresses of every contract on the blockchain. It also has a cell where we place commands, `<eeiK>`, which will be used for passing control.

2.3.2 A General Template for Making a Wasm Embedding

As we have mentioned, Wasm is made to be embedded. In designing KEwasm we wanted to come up with a general template for embedding Wasm in a host environment while not needing to modify the host semantics or the Wasm semantics in any way. That means we want to be able to pass control cleanly between the Wasm engine and the embedder, without having a situation where both could proceed

a typical transaction February 2020 is priced at around 1-10 GWei (10^{-9} ether). Thus, creating a contract with 1 page of storage would come out to between 8 and 80 USD. There is, however, another solution. Just like in KWasm, memory could be treated as sparse, and only non-zero values stored. Still, it could become quite expensive, since memory is where call-data gets stored and many operations are performed on data that does not fit in the built-in data types. If memory was used as persistent storage there would be several problems, such as the caller would have to pay for their transient storage—which is at least the call-data—as if it was persistent, contracts would need to have extra logic to separate persistent data from temporary, or there would be extra costs associated with wiping memory after each invocation. While a scheme may be implemented in the future that uses Wasm linear memory as persistent—perhaps allowing more than one memory per module—in KEWasm we have chosen the semantics that linear memory is wiped after each transaction.

[12]The interested reader could try it in Python with the following one-liner: `sha3.keccak_256 ("balance(address):(uint64)".encode()).digest()[0:4]`. The result should be the hexadecimal bytes `99 93 02 1a`. Make sure the `pysha3` package from `pip` is installed first.

[13]In the current, prototypical version of Ewasm, that is.

concurrently. Since $\mathbb{K}$ is specifically designed to make concurrent semantics easy to specify, this requires some care.

2.3.2.1 Dealing With Host Functions in KWasm

We extend the Wasm syntax with a new sort. We then resolve the function imports by creating regular functions in Wasm, with a single `HostCall` in the function body.

Listing 2.6: Semantics of Ewasm host calls.

```
1     syntax Instr ::= HostCall
2  // ------------------------
3 rule <k> ( import MODNAME FNAME (func OID:OptionalId TUSE:
     TypeUse) )
4        => ( func OID TUSE .LocalDecls #eeiFunction(FNAME) .
            Instrs )
5        ...
6        </k>
7   requires MODNAME ==K #ethereumModule
```

Here, `#ethereumModule` is a macro for the string `"ethereum"` and `#eeiFunction` is a mapping between import names as seen in listing **??** and `HostCalls`. Each `HostCall` is a single terminal. For each terminal, we define a chain of transitions that triggers the corresponding execution on the host side and then wait for the result, before consuming the result.

2.3.2.2 Creating a Wasm-Host Boundary

With the semantics for Wasm and the EEI specified separately, all that remains to complete EWasm is to compose the two semantics and specify the interaction at the boundary.

The following configuration specifies KEwasm:

```
1 configuration
2   <ewasm>
3     <eei/>
4     <wasm/>
5     <paramstack> .ParamStack </paramstack>
6   </ewasm>
```

The `<paramstack>` cell helps with translating between the two worlds. In the EEI values are encoded as integers, implicitly 256-bit words. In Wasm words are at most 8 bytes. Linear memory serves as the glue between the worlds.

The following is an example of how KEWasm specifies the interaction with the host functions. The example is of calling `$getCaller`, which returns the address of the account that initiated the transaction. Addresses are 20 bytes long and get returned to memory.

```
1 syntax HostCall ::= "eei.getCaller"
2 // ----------------------------------
3 rule <k> eei.getCaller => #waiting(eei.getCaller) ... </k>
```

```
4       <eeiK> . => EEI.getCaller </eeiK>
5
6   rule <k> #waiting(eei.getCaller) => #storeEeiResult(RESULTPTR
        , 20, ADDR) ... </k>
7       <locals> 0 |-> <i32> RESULTPTR </locals>
8       <eeiK> #result(ADDR) => . </eeiK>
```

The function `#storeEeiResult` is a helper that expands to a sequence of `i32.store8` calls, one for each byte to be stored. In the above case, the address will be stored starting at `RESULTPTR`, continuing for 20 bytes, and the value to be stored will be that returned in `ADDR`.

An example of how `<paramstack>` helps with calls is the `$storageStore` function. The boundary between the EEI and Wasm is given below:

```
1   syntax HostCall ::= "eei.storageStore"
2   // ---------------------------------------
3   rule <k> eei.storageStore => #gatherParams(eei.storageStore,
        (INDEXPTR, 32) (VALUEPTR, 32)) ... </k>
4       <locals>
5         0 |-> <i32> INDEXPTR
6         1 |-> <i32> VALUEPTR
7       </locals>
8
9   rule <k> #gatheredCall(eei.storageStore) => #waiting(eei.
        storageStore) ... </k>
10      <paramstack> VALUE : INDEX : .ParamStack => .ParamStack
            </paramstack>
11      <eeiK> . => EEI.setAccountStorage INDEX VALUE </eeiK>
12
13  rule <k> #waiting(eei.storageStore) => . ... </k>
14      <eeiK> . </eeiK>
```

In this call, there is an intermediate step before the EEI takes control. That step gathers parameters from linear memory. `#gatherParams` takes a continuation in the form of the host call that initiated the gathering, and a list of index-length pairs. It then converts the corresponding memory locations into 32-byte integers and puts them on the `<paramstack>` cell, before putting `#gatheredCall(CONTINUATION)` at the top of the `<k>` cell. The middling function then consumes the parameter stack and the continuation and passes control to the EEI.

With similar functions for each of the necessary host calls, we have a prototype Ewasm embedding powerful enough to verify the WRC20 contract we have in mind. The remaining host functions are left unimplemented until they are needed for verification of some contract, or until the final Ewasm hot interface has been decided.

2.3.2.3 Testing The Ewasm Semantics

In addition to unit testing the host calls, we add unit tests for the WRC20. We use two different versions, one for storing the balances in reversed-byte order for fast retrieval when calling the `balance` function, and one which stores the balances in as

regular integers, for doing fast transfers. The unit tests do a single transfer, checking that the balances are correct before and after. The test for the fast-balances version can be found at `https://github.com/kframework/ewasm-semantics/blob/master/tests/simple/wrc20_fast_get_balance.wast`, and the test for the fast-transfers version can be found at `https://github.com/kframework/ewasm-semantics/blob/master/tests/simple/wrc20_fast_transfer.wast`.

3

Part 2: KWasm for Deductive Program Verification

3.1 From Semantics to Proofs

As described in Sect. 1.7.4, a $\mathbb{K}$ semantics can be read and understood as a computational transition system specifying an interpreter. But it can also be understood as a logic theory under which we can prove properties about programs. In addition to using a `requires` clause, these rules often use an `ensures` clause to state postconditions.

We could express our semantics in mathematical notation the following way: Call the set of rewrite rules in KWasm Σ. These are axioms. Call the full theory (all theorems which are provable from the axioms) T. Of course, $\Sigma \subseteq T$. If a rule $L \Rightarrow R$ requires $C \in T$, that means that given conditions C, any program that matches L will eventually rewrite to R, or never terminate.

To use the $\mathbb{K}$ framework for deductive program verification, one writes proof obligations which the $\mathbb{K}$ prover tries to prove or disprove belong to T. A proof obligation in $\mathbb{K}$ is specified exactly like a regular semantic rule, $L \Rightarrow R$ requires C. Just like in a semantic rule, the values mentioned in the obligation may be symbolic. A set of these proof obligations, S, is called a $\mathbb{K}$ *spec* . As an example, here we declare a semantics of the language `foo` with only one production and two transition rules. In the second module `FOO-SPEC`, we put a proof obligation.

Listing 3.1: A small proof in $\mathbb{K}$.

```
1  module FOO
2    imports INT
3
4    configuration
5      <k> $PGM:P </k>
6
7      syntax P ::= "foo" Int Int
8   // ------------------------
9      rule [a]: <k> foo X Y => foo Y X ... </k>
10     rule [b]: <k> foo X Y => X +Int Y +Int 1 ... </k>
11
12 endmodule
13
14 module FOO-SPEC
15   imports FOO
```

```
16
17      rule <k> foo X Y => ?Z ... </k>
18        ensures ?Z >Int X +Int Y
19
20  endmodule
```

The proof obligation is a theorem iff starting in the configuration `<k> foo X Y ... </k>` (with all cells except the `<k>` cell unspecified), all paths either

1. do not terminate, i.e. will perform rewrites forever, or
2. end up with an integer on top of the `<k>` cell, everything else that was initially following `foo X Y`, indicated by `...`, as well as the rest of the configuration was left unchanged, and the final integer is larger than the sum of `X` and `Y`.

The `?` is an existential quantifier, saying that there is such an integer, rather than that this claim works for any integer.

This spec is indeed provable. All paths which eventually apply rule `b` would match with the right-hand side of the proof obligation. The path which applies rule `a` forever will never terminate. This is enough for the spec to pass.

3.2 A Very Simple Proof

Below is a simple spec which asserts that copying the value of a local variable to the stack with `local.get` and then writing that value back to the same variable with `local.set`

1. terminates normally, as expressed by the whole program rewriting to `.`, and
2. produces no other changes in the state since there are no other rewrites.

Listing 3.2: A spec for reading from and writing to the value of local variables.

```
1   module LOCALS-SPEC
2       imports WASM-TEXT
3       imports KWASM-LEMMAS
4
5       rule <k> (local.get X:Int) (local.set X:Int) => . ... </k
            >
6             <locals>
7               X |-> < ITYPE > VAL
8             </locals>
9
10  endmodule
```

The program in the `<k>` cell is simple to verify because, during the course of its evaluation, only one semantic rule ever applies at a time. We will go over the process in detail below.

The following are the relevant rules in KWasm for this proof:

Listing 3.3: The heating and unfolding rules in KWasm.

```
1   rule <k> (S:Stmt SS) => S ~> SS ... </k>
2     requires SS =/=K .Stmts
3
```

```
4  rule <k> ( PI:PlainInstr ):FoldedInstr => PI ... </k>
5
6  rule <k> local.get I:Int => . ... </k>
7       <valstack> VALSTACK => VALUE : VALSTACK </valstack>
8       <locals> ... I |-> VALUE ... </locals>
9
10 rule <k> local.set I:Int => . ... </k>
11      <valstack> VALUE : VALSTACK => VALSTACK </valstack>
12      <locals> ... I |-> (_ => VALUE) ... </locals>
```

The initial configuration is exactly the left-hand side of the proof obligation. First, the heating rule on line 1 applies, followed by the unfolding rule on line 4. Now the configuration has become the following:

```
<k> local.get X ~> (local.set X) ... </k>
<valstack> VALSTACK </valstack>
<locals>
  X |-> < ITYPE > VAL
</locals>
```

Here `VALSTACK` is whatever the stack contained before.

Next, the rule for `local.get` on line 6 applies[1] This produces a `.` on top of the `<k>` cell, which is removed. Conceptually, $\mathbb{K}$ has a built-in rule that looks like `rule . ~> THEN => THEN`, and the `...` in a rule can be replaced by `~> THEN` (as long as the name `THEN` is free, of course). Then, unfolding applies again. This gives us a new configuration:

```
<k> local.set X:Int ... </k>
<valstack> < ITYPE > VAL : VALSTACK </valstack>
<locals>
  X |-> < ITYPE > VAL
</locals>
```

Lastly the rule for `local.set`[2] on line 10 applies: This gives the final configuration, which matches the right-hand side of the configuration:

```
<k> . ... </k>
<valstack> VALSTACK </valstack>
<locals>
  X |-> < ITYPE > VAL
</locals>
```

In this simple case, we were able to simply state how a program would terminate[3] and leave the state unchanged, and the prover could infer it for us. Indeed, in making this example, the specification above was written and proved on the first try. The proving process is not always so straightforward, however.

[1] The rule is paraphrased here, it actually is slightly more complex to deal with identifiers.

[2] Again, paraphrased.

[3] We have not, however, produced a proof that this program must terminate—we have only concluded partial correctness.

3.3 A Proof With Heap Reasoning

Some proofs require that we further specify our intended semantics and encode the invariants of the transition system. As an example, we take the exact analog of our previous proof. Only this time, instead of modifying local variables we are modifying heap storage.

`#inUnsignedRange` captures the invariants that all integer values, once passed through an `ITYPE.const`, will be represented by their corresponding unsigned value, regardless of signed representation. I.e., any integer value in the state, except at some points in the `<k>` cell, is assumed to be in $\mathbb{Z}_{\text{ITYPE}}$, and the representative is chosen to be the value of the class in the range 0 to $2^{|\text{ITYPE}|-1}$. An invariant the semantics have been designed to maintain is that of `#isByteMap`.[4]

Listing 3.4: Specification for a simple memory property.

```
1  module MEMORY-SPEC
2      imports WASM-TEXT
3      imports KWASM-LEMMAS
4
5      rule <k> (i32:IValType.store (i32.const ADDR) (i32.load (
           i32.const ADDR)):Instr):Instr => . ... </k>
6           <curModIdx> CUR </curModIdx>
7           <moduleInst>
8             <modIdx> CUR </modIdx>
9             <memAddrs> 0 |-> MEMADDR </memAddrs>
10            ...
11          </moduleInst>
12          <memInst>
13            <mAddr> MEMADDR </mAddr>
14            <msize> SIZE </msize>
15            <mdata> BM   </mdata>
16            ...
17          </memInst>
18        requires #chop(<i32> ADDR) ==K <i32> EA
19         andBool EA +Int #numBytes(i32) <=Int SIZE *Int #
              pageSize()
20         andBool #isByteMap(BM)
21
22 endmodule
```

The prover will not deduce that this spec holds. The reason is that storing to and reading from memory is more complicated than storing local values. Wasm uses byte-addressable storage. That is, when a value is stored to memory it is spliced into bytes. When a value is read from memory, the bytes are assembled into an integer value. Conceptually, the `load` operation will push the following onto the stack:

[4]Both the invariants that integers anywhere but the `<k>` cell are in a specific range, and that the memory is indeed a valid byte map, has yet to be proven.

$$val = bm[addr] + (bm[addr+1] + (bm[addr+2] + bm[addr+3] * 256) * 256) * 256 \tag{3.1}$$

The `store` operation pops a value off the stack, and stores the following sequence of bytes to memory:

$$bm[addr] := val \mod 256 \tag{3.2}$$
$$bm[addr+1] := (val/256) \mod 256 \tag{3.3}$$
$$bm[addr+2] := (val/256^2) \mod 256 \tag{3.4}$$
$$bm[addr+3] := (val/256^3) \mod 256 \tag{3.5}$$

If we plug *val* from Eq. 3.1 into the assignments 3.2—3.5 it becomes clear that the modulus and division operators will cancel out exactly so all we are doing is writing the values in each address back to their place in memory.

This type of reasoning presents a challenge for the $\mathbb{K}$ prover using the current semantics. The semantics uses pure helper functions, `#setRange` and `#getRange` for writing to and reading from the byte map. These functions expand to a series of `#set` and `#get`, that do the obvious[5].

There is only ever one rule which applies at each step in this proof, so there is no branching. In the end, Z3 is queried and is tasked with deciding whether the assignments 3.2—3.5 really are identity operations—i.e., that each byte is just written back to its original position. A human verifier with a pen and paper can easily see this. However, Z3 can not reason about these functions in the way we would like without giving full definitions in Z3 of the setter and getter functions themselves. Since the getting and setting happen at the $\mathbb{K}$ level while the arithmetic reasoning happens at the Z3 level, we are stuck. We can remedy this by either extending Z3's reasoning capabilities, or the $\mathbb{K}$ framework's. In this case, we chose to extend the $\mathbb{K}$ framework. We add the following lemmas, which should obviously hold in integer and modular arithmetic[6].

Listing 3.5: Five lemmas about arithmetic for heap reasoning.

```
1 rule (X *Int N +Int Y) modInt N => Y modInt N
2 rule (Y +Int X *Int N) modInt N => Y modInt N
3
4 rule 0 +Int X => X
5 rule X +Int 0 => X
6
7 rule (Y +Int X *Int N) /Int N => (Y /Int N) +Int X
```

[5]Actually, there is one non-obvious part of each function: when the stored value is 0, that is represented by no entry. The two functions respect that by erasing 0-valued entries and interpreting an empty entry as 0, respectively.

[6]N.B. that we cannot use the distributive property of division. It holds over the rational, real and complex numbers (which are fields w.r.t. addition and multiplication), not over the integers (which is a ring).

Together, they help eliminate the expressions for assignment to:

$$
\begin{array}{rlll}
bm[addr] := bm[addr] & & \bmod 256 & \\
bm[addr+1] := bm[addr] & /256 & \bmod 256 + bm[addr+1] & \bmod 256 \\
bm[addr+2] := bm[addr] & /256^2 & \bmod 256 + bm[addr+1]/256 & \bmod 256 \\
\quad + bm[addr+2] & & \bmod 256 & \\
bm[addr+3] := bm[addr] & /256^3 & \bmod 256 + bm[addr+1]/256^2 & \bmod 256 \\
\quad + bm[addr+2] & /256 & \bmod 256 + bm[addr+3] & \bmod 256
\end{array}
$$

We can now make use of the invariant that we claim to maintain for the byte map. We add the following two lemmas:

Listing 3.6: Two lemmas about `#get` for heap reasoning.

```
1 rule #get(BMAP, IDX) modInt 256 => #get(BMAP, IDX)
2   requires #isByteMap(BMAP)
3 rule #get(BMAP, IDX) /Int   256 => 0
4   requires #isByteMap(BMAP)
```

They state that as long as a byte map maintains its intended invariant—that all values are integers from 0 to 255 inclusive—we may discard the modulus on the values and the division amount to zeroing them. The lemma in itself is self-evident since it assumes the byte map maintains the invariant. The claim that our semantics maintain the invariant that memory is always a byte map is, at present, a conjecture.

With the lemmas from listings 3.5 and 3.6 added to our axioms, the proof goes through.

3.3.1 Using a Symbolic Type

If we want to make a proof that uses a symbolic type, rather than `i32` or `i64`, matters become more complicated. Without knowing the type, `#setRange` and `#getRange` will receive a symbolic `WIDTH` argument, and not be able to expand.

To make a proof like that go through, we introduce a more specialized idempotence lemma. But rather than including it in the set of manual axioms for all our verification, we can apply it locally where it is needed.

Listing 3.7: A lemma over `#setRange` and `#getRange`.

```
1 require "kwasm-lemmas.k"
2
3 module MEMORY-SYMBOLIC-TYPE-LEMMAS
4   imports KWASM-LEMMAS    // The set of lemmas included in
      every proof.
5
6   rule #setRange(BM, EA, #getRange(BM, EA, WIDTH), WIDTH) =>
      BM
7
8 endmodule
9
```

```
10 module MEMORY-SYMBOLIC-TYPE-SPEC
11     imports WASM-TEXT
12     imports MEMORY-SYMBOLIC-TYPE-LEMMAS
13
14     rule <k> (ITYPE:IValType.store (i32.const ADDR) (ITYPE.
           load (i32.const ADDR)):Instr):Instr => . ... </k>
15
16 ...
```

By invoking the $\mathbb{K}$ prover with the option `--def-module MEMORY-SYMBOLIC-TYPE-LEMMAS` instead of the usual `--def-module KWASM-LEMMAS`, the prover will include this new lemma into its axioms, and the proof will go through.

3.4 Helping the $\mathbb{K}$ Prover With Inductive Reasoning

The following Wasm program calculates the arithmetic sum $\sum_{i=0}^{N} i$. That is, assuming the first local variable (index 0) contains N and the second local variable contains 0.

Listing 3.8: A Wasm program for summing the numbers 1 to N.

```
1  block
2      ( loop
3          (local.get 0)
4          (local.get 1)
5          (i32.add)
6          (local.set 1)
7          (local.get 0)
8          (i32.const 1)
9          (i32.sub)
10         (local.tee 0)
11         (i32.eqz)
12         (br_if 1)
13         (br 0)
14     )
15 end
```

The result will be that the sum, known to be $N(N-1)/2$, is in the second local variable at the end of execution. (Under which circumstances will this work correctly?)

We aim to prove this using the following claim, where the above program is in the `<k>` cell. The proof obligation is to show that

1. the program terminates (rewrites to `.`), and
2. the locals are updated according to our expectations of the program's behavior.

We assume the standard invariants hold for integer values we encounter, namely them being in range correct range. This includes the final result—we assume no overflow occurs. The looping code is also written so that it assumes the first local

is not 0 at the outset. The subtraction on line 9 occurs before the 0-check on line 11. These three assumptions are encoded in the `requires`-clause below.

Listing 3.9: Spec with the main proof obligation for the arithmetic sum program.

```
1  module LOOPS-SPEC
2      imports WASM-TEXT
3      imports KWASM-LEMMAS
4
5      // Main claim.
6      rule <k> block .TypeDecls
7                   ( loop .TypeDecls
8                       (local.get 0)
9                       (local.get 1)
10                      (i32.add)
11                      (local.set 1)
12                      (local.get 0)
13                      (i32.const 1)
14                      (i32.sub)
15                      (local.tee 0)
16                      (i32.eqz)
17                      (br_if 1)
18                      (br 0)
19                  )
20               end
21            => .
22            ...
23           </k>
24           <locals>
25             0 |-> < i32 > (N => 0)
26             1 |-> < i32 > (0 => (N *Int (N +Int 1)) /Int 2)
27           </locals>
28       requires #inUnsignedRange(i32, N)
29        andBool #inUnsignedRange(i32, ((N *Int (N +Int 1)) /
             Int 2))
30        andBool N >Int 0
31
32 endmodule
```

However, the prover does not succeed in proving this claim. After 120 steps, KLab shows the following view of the prover's execution:

```
- 0            (((N - 1) == 0) == false)
| - 0          ((((N - 1) - 1) == 0) == false)
| | - 0        (((((N - 1) - 1) - 1) == 0) == false)
| | | - 0      ((((((N - 1) - 1) - 1) - 1) == 0) == false)
| | | _ 1 *    (((((N - 1) - 1) - 1) - 1) == 0)
| | _ 1 *      ((((N - 1) - 1) - 1) == 0)
| _ 1 *        (((N - 1) - 1) == 0)
_ 1 *          ((N - 1) == 0)
```

It seems the prover can only prove the ground cases, where N gets a concrete value because it is defined by an equation. For example, when $(N-1)-1=0$ is true we have $N=2$. However, when all we know is that $N-1\neq 0$ and $(N-1)-1\neq 0$, the prover fails to conclude the claim. Instead, it keeps computing in another loop iteration.

The prover can do inductive reasoning through the use of reachability logic's circularity proof rule[36, Sect. 3]. To help the prover along and make it succeed in doing inductive reasoning, we show it what situation it will encounter over and over in its attempt on the main claim. This differs slightly from the typical task of identifying a loop invariant, where one specifies something that will always be true at the end of *each iteration*. Instead, we identify the beginning of the loop and make a claim from the beginning of the loop until the end of *the program*.

To help the prover in this instance, we give the following lemma.

Listing 3.10: The proof-obligation for inductive reasoning.

```
1     // Lemma
2     rule <k> br 0
3               ~> label // Loop label.
4                    [ .ValTypes ]
5                    { loop .TypeDecls
6                        (local.get 0)
7                        (local.get 1)
8                        (i32.add)
9                        (local.set 1)
10                       (local.get 0)
11                       (i32.const 1)
12                       (i32.sub)
13                       (local.tee 0)
14                       (i32.eqz)
15                       (br_if 1)
16                       (br 0)
17                   end
18                }
19                .ValStack
20              ~> label [ .ValTypes ] {.Instrs } STACK // Block
                    label.
21            => .
22               ...
23          </k>
24          <valstack> _ => STACK </valstack>
25          <labelDepth> D => D -Int 2 </labelDepth>
26          <locals>
27            0 |-> < i32 > (I => 0)
28            1 |-> < i32 > (X => X +Int ((I *Int (I +Int 1)) /
                 Int 2))
29          </locals>
30        requires #inUnsignedRange(i32, I)
31         andBool I >Int 0
32         andBool #inUnsignedRange(i32, X +Int I)
```

```
33          andBool #inUnsignedRange(i32, X +Int ((I *Int (I +Int
               1)) /Int 2))
```

The `<k>` cell contains exactly what it will contain at each branching point in the proof. The rest of the configuration expresses the expected state transition. In particular, the `<locals>` cell is specified to rewrite exactly as it would in the main claim, except now it must start with some symbolic value in the second local variable, instead of 0.

The prover succeeds in proving the lemma through inductive reasoning, and proving the main claim then becomes trivial by executing the program until the loop and then using the lemma. In general, a fruitful approach is to simply examine the exact state of the symbolic configuration at any branching point in the loop, and considering how we expect it to have been transformed by the end of the program.

3.5 WRC20: Specifying and Proving Correct the `i64.reverse_bytes` Function[7]

WRC20 is a Wasm version of an ERC20. In essence, the ERC20 standard specifies an interface for smart contracts where Ethereum addresses can own a number of tokens, transfer them, and give other addresses permission to spend tokens on their behalf.

The WRC20 program can be found here. It is simpler than an ERC20: it only has a function for transferring (the caller's) funds to another address and a function for querying the balance of any address. Keep in mind also that Ewasm is part of Ethereum 2.0, phase 2. It is still a work in progress, so exactly what Ewasm will look like is unclear. This is based on an early work-in-progress specification of Ewasm.

In the end, we want to verify the behavior of the two external functions, `transfer` and `getBalance`. This will require us to use the Ewasm embedding.

For now, we will focus on a helper function: `$i64.reverse_bytes`. This is a pure Wasm function that takes an `i64` as a parameter and returns the `i64` you get by reversing the order of the bytes.

Listing 3.11: `$i64.reverse_bytes` Wasm code. Copyright Paul Dworzanski et al., licensed under GNU General Public License v. 3.

```
1  (func $i64.reverse_bytes (param i64) (result i64)
2    (local i64 i64)     ;;iter variable, val to return
3    block
4      loop
5        local.get 1     ;;iter variable
6        i64.const 8
7        i64.ge_u
8        br_if 1
```

[7]Parts of this section has been previously published by the author in a blog post, available here:
`https://medium.com/dlabvc/verifying-wasm-functions-part-2-i64-reverse-bytes-3590aedaa3c0`.

```
9           local.get 0      ;;original
10          i64.const 56     ;;shift left
11          local.get 1
12          i64.const 8
13          i64.mul
14          i64.sub
15          i64.shl
16          i64.const 56     ;;shift right
17          i64.shr_u
18          i64.const 56     ;;shift left
19          i64.const 8
20          local.get 1
21          i64.mul
22          i64.sub
23          i64.shl
24          local.get 2      ;;update
25          i64.add
26          local.set 2
27          local.get 1      ;;iter+=1
28          i64.const 1
29          i64.add
30          local.set 1
31          br 0
32        end
33      end
34      local.get 2
35    )
```

Suppose the input consists of the following bytes:

```
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
```

Then the output becomes:

```
x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
```

This function exists because Wasm stores numbers to memory as little-endian, while Ethereum uses a big-endian representation in the call data. So the contract must reverse all integer balances: in `transfer`, it must reverse the amount to be sent, and in `getBalance` it must reverse the result before returning.

3.5.1 The Proof Obligation

Without further ado, here is what we are going to prove[8]:

Listing 3.12: The spec for `$i64.reverse_bytes` function.

[8]We should note that this is slightly different from the lemma we will need in the end when verifying the `transfer` and `getBalance` functions. To make this spec useful, we need to make sure the starting state of the spec matches the state of the program when `transfer` and `getBalance` calls `$i64.reverse_bytes`. If we do that, the prover will be able to go: "Aha! This state corresponds exactly to something I've proven, I can just jump to the conclusion!" But the above version makes for a nice presentation.

```
1     rule <k> #wrc20ReverseBytes // A macro. Expands to the
          function defintion.
2          ~> (i32.const ADDR)
3             (i32.const ADDR)
4             (i64.load)
5             (invoke NEXTADDR) // Invoke is an internal Wasm
                 command, similar to `call`.
6             (i64.store)
7          => .
8             ...
9         </k>
10        <curModIdx> CUR </curModIdx>
11        <moduleInst>
12          <modIdx> CUR </modIdx>
13          <memAddrs> 0 |-> MEMADDR </memAddrs>
14          <types> TYPES => _ </types>
                                                  /* These
                 five state changes  */
15          <nextTypeIdx> NEXTTYPEIDX => NEXTTYPEIDX +Int 1 </
                 nextTypeIdx>  /* are due to the fact that  */
16          <funcIds> _ => _ </funcIds>
                                                  /* we
                 declare a new function */
17          <funcAddrs> _ => _ </funcAddrs>
                                               /* in the first
                 step of the  */
18          <nextFuncIdx> NEXTFUNCIDX => NEXTFUNCIDX +Int 1 </
                 nextFuncIdx>  /* specification.            */
19          ...
20        </moduleInst>
21        <funcs> _ => _ </funcs>
                                                /* So is this
              change. */
22        <nextFuncAddr> NEXTADDR => NEXTADDR +Int 1 </
             nextFuncAddr>   /* And this one.      */
23        <memInst>
24          <mAddr> MEMADDR  </mAddr>
25          <msize> SIZE     </msize>
26          <mdata> BM => ?BM' </mdata>
27          ...
28        </memInst>
29      requires notBool unnameFuncType(asFuncType(#
           wrc20ReverseBytesTypeDecls)) in values(TYPES)
30       andBool #isByteMap(BM)
31       andBool #inUnsignedRange(i64, X)
32       andBool #inUnsignedRange(i32, ADDR)
33       andBool ADDR +Int #numBytes(i64) <=Int SIZE *Int #
            pageSize()
34      ensures  #get(BM, ADDR +Int 0) ==Int #get(?BM', ADDR +
```

```
          Int 7 )
35        andBool #get(BM, ADDR +Int 1) ==Int #get(?BM', ADDR +
             Int 6 )
36        andBool #get(BM, ADDR +Int 2) ==Int #get(?BM', ADDR +
             Int 5 )
37        andBool #get(BM, ADDR +Int 3) ==Int #get(?BM', ADDR +
             Int 4 )
38        andBool #get(BM, ADDR +Int 4) ==Int #get(?BM', ADDR +
             Int 3 )
39        andBool #get(BM, ADDR +Int 5) ==Int #get(?BM', ADDR +
             Int 2 )
40        andBool #get(BM, ADDR +Int 6) ==Int #get(?BM', ADDR +
             Int 1 )
41        andBool #get(BM, ADDR +Int 7) ==Int #get(?BM', ADDR +
             Int 0 )
```

The interesting parts are:

- the `<k>` cell,
- the `<memInst>` cell group, and
- the pre- and postconditions, `requires` and `ensures`.

3.5.1.1 The <k> Cell

Here we first declare the function, which we have saved, in pre-parsed form as a macro. This will store the function in the state, which means several updates will happen. A new type and a new function address pointer get added to the module instance, and a new function gets added to the world of functions (`<funcs>`). After that (remember, `~>` should be read as "and then"), we run a few Wasm instructions that load 8 bytes from memory, invokes the `i64.reverese_bytes` function, and stores the result back to the same address.

3.5.1.2 The <memInst> Cell

The `<memInst>` cell simply states that there is a memory with address `MEMADDR`, the same as int `<memAddrs>` in `<moduleInst>`, which makes this the memory belonging to the module we are currently executing in. We also state that the memory gets rewritten, from `BM` to `?BM'`. Every part of the state that we do not state gets rewritten will be assumed to stay the same. So if we omitted this rewrite, the proof obligation would be stating that the memory does not change at all. We also take the `SIZE` of the memory into account, given in the number of pages (each page is 64 KiB).

3.5.1.3 The Pre- and Postconditions

The `requires` and `ensures` sections say what we assume to be true when at the outset of the proof and what we need to prove at the end of the proof. Note that some pre- and postconditions are expressed in the rewrite rules themselves, such as the value in `<memAddrs>` of the current module matching the `<mAddr>` of the `<memInst>` that gets changed (precondition) and that the program in the `<k>` gets consumed

(postcondition). The `requires` and `ensures` clauses are simply for stating facts that we cannot express directly in the rewrite.

The first 4 requirements are really boilerplate relevant to the technicalities of the semantics. The first states that the type of the `i64.reverse_bytes` function has not already been declared.[9] The second, third and fourth rules all make sure that integers, whether constants or stored in the byte map, are in the allowed range. Without these assumptions, the prover assumes the values are unbounded integers.

The final clause in the `requires` section states that our memory accesses are within bounds. This is why we need to know the size (`SIZE`) of the memory. A separate (but less interesting) proof would show that the function causes a `trap` if this precondition is violated.

The `ensures` section is straightforward. We are simply asking the prover to ensure that the final memory has correctly flipped the bytes.

3.5.2 Helping the Prover Along

Simply stating the above proof obligation and giving it to the prover will result in inconclusive results. The prover will fail having neither proven or disproved the claim.

One reason for this failure is that under the hood, there is a good deal of modular arithmetic going on. This happens when we transition from the bytes in memory to integer values, and back. $\mathbb{K}$ does not (yet) have much support for reasoning about modular arithmetic. This presents an excellent opportunity to add that support. We will extend the set of axioms $\mathbb{K}$ knows, triple-checking each (so that we do not introduce unsoundness), directed by the places where the prover gets stuck. $\mathbb{K}$ also supports adding lemmas to Z3 to reason about, which is in some ways simpler. We discuss the trade-offs between adding $\mathbb{K}$ and Z3 lemmas in Sec. 4.2, but for now it suffices to say we prefer adding lemmas to $\mathbb{K}$ whenever possible.

3.5.2.1 Axiom Engineering: Avoiding Infinite Rewrites

The new axioms need to be designed with care. Apart from making sure that they are correct, we want to ensure that they are simplifications: that is, they reduce the state. This can be by making expressions or values smaller. The reason is that we want to ensure we cannot get explosive growth. The prover will try to apply all axioms whenever possible. Therefore, if we were to add statements like `X +Int Y => Y +Int X`, which does not reduce the state, we are in trouble—even though this is perfectly sound.

If we were to add a rule that says `X +Int Y => Y +Int X`, then the prover will apply this rewrite wherever it can, over every addition it sees. But once it is done, there are still just as many additions in the state, and the prover will again apply this rewrite on all additions it sees.

[9] This is a somewhat arbitrary choice. There is a semantic rule which declares the function type if it is not already present. There are some technicalities associated with declaring and looking up functions. By letting the prover go through those steps, it can construct the state of the `TYPES` the way the semantics specifies. This way, the proof becomes more robust (and readable) than if we wrote out the expected state of the types directly in the proof.

To avoid this we follow a few general principles when engineering our axioms:

Every rule must do at least one of the following:

1. Reduce the number of `modInt`, `>>Int` or `<<Int` constructs, or push `<<Int` up the parse tree, or `>>Int` down the parse tree[10]. This may alter the expression in any other way, by making it larger or increase the values of integers.

2. Make the expression strictly smaller without introducing or moving new `modInt` , `>>Int` or `<<Int` constructs.

3. Make some integer in the expression strictly closer to 0 without introducing or moving any `modInt`, `>>Int` or `<<Int`, making the expression larger[11] or making some other integer farther from 0.

With these rules, we can avoid infinite rewrites. The only way for the expression sizes or integer sizes to grow is by removing a `modInt`, `<<Int` or `>>Int`, or pushing them towards some end of the parse tree. There is no way to introduce more of these operations, and the shifts will, if nothing else, eventually reach either the root or leaves. Therefore, the first rule can apply only a finite number of times. This puts a hard upper limit on how large any given expression can grow through axiom application. The third rule can also only be applied a finite number of times since the expression sizes are bounded from above. The fourth rule can also not apply infinitely often since there can only be a finite number of integers in a finite expression, and once these all go to 0, the rule can no longer apply.

This trick is called making a "lexicographic product". This is a product of three orders. You can think of it as each expression being represented by a tuple of three numbers, `(b, e, n)`, for "bits", "arithmetic" and "numbers". `b` is the number of `modInt` expressions plus the distances of all `>>Int` from their farthest leaf plus the distances of `<<Int` constructors from the root[12]in the expression, `e` is the total number of constructors (nodes in the parse tree), and `n` is the sum of all integers[13] in the expression. Now, we introduce a total ordering over all expressions, by first comparing their `b` numbers, then their `e` numbers, then their `n` numbers. It is like comparing two words to figure out which goes first in a lexicon (hence the name "lexicographic order"): first look at the initial letter, then the second, and so on, until they are different, at which point you know which comes first. So $(3, 4, 1) < (3, 4, 2)$ and $(1, 1000, 1000) < (2, 0, 0)$. Now revisit our rules for adding axioms.

- When rule 1 applies, the `b` number of the expression decreases, but the others may increase.
- When rule 2 applies, the `a` number decreases, the `b` number stays the same, and the `n` number may go up.
- And when rule 3 applies, the `n` number decreases, and `b` and `a` stay the same.

The result is that for every rule application, we get a "smaller" expression than we had before, and no expression can become smaller than (0, 0, 0), which gives a

[10]Further up or down means closer to the root, or closer to the *farthest* leaf below them.

[11]The expression may grow by a single constructor, such as a `-Int` with a constant right operand, as long as the result is strictly closer to 0. This imposes some extra restrictions on how expressions may shrink in other axioms.

[12]Really the sum of the absolute values, but since negative integers occur only in very specific places in the semantics, the distinction is mostly irrelevant.

[13]Really the sum of the absolute values, but since negative integers occur only in very specific places in the semantics, the distinction is mostly irrelevant.

hard limit on how many rewrites can apply.

This kind of careful engineering is necessary for axioms we want to add to $\mathbb{K}$'s reasoning capabilities for all programs. For verifying specific languages or programs we may get away with being a little less rigor. However, it is good practice to try to ensure that your axioms are not causing your prover to loop forever[14].

3.5.2.2 Adding New Axioms

When we conduct the proof without adding anything to the semantics, the prover manages to symbolically run the program to termination. The contents of the `<k>` cell are gone, rewrites have happened in all the expected places.

However, the memory array in `<mdata>`, `BM`, will have been replaced by a behemoth of symbolic expression, see appendix A. The reason for the size is that the function returns the following large expression, which is the reversed `i64`, and repeats it 8 times, once for each byte that gets inserted into memory.

Listing 3.13: The expression for integer interpretation of the reversed bytes

```
1 ( ( ( #getRange ( BM , ADDR , 8 ) <<Int 0 ) modInt
    18446744073709551616 >>Int 56 <<Int 0 ) modInt
    18446744073709551616 +Int ( ( ( #getRange ( BM , ADDR , 8
    ) <<Int 8 ) modInt 18446744073709551616 >>Int 56 <<Int 8 )
     modInt 1844674407370 9551616 +Int ( ( ( #getRange ( BM ,
    ADDR , 8 ) <<Int 16 ) modInt 18446744073709551616 >>Int 56
     <<Int 16 ) modInt 18446744073709551616 +Int ( ( ( #
    getRange ( BM , ADDR , 8 ) <<Int 24 ) modInt
    18446744073709551616 >>Int 56 <<Int 24 ) modInt
    18446744073709551616 +Int ( ( ( #getRange ( BM , ADDR , 8
    ) <<Int 32 ) modInt 18446744073709551616 >>Int 56 <<Int 32
     ) modInt 18446744073709551616 +Int ( ( ( #getRange ( BM ,
     ADDR , 8 ) <<Int 40 ) modInt 18446744073709551616 >>Int
    56 <<Int 40 ) modInt 18446744073709551616 +Int ( (( #
    getRange ( BM , ADDR , 8 ) <<Int 48 ) modInt
    18446744073709551616 >>Int 56 <<Int 48 ) modInt
    18446744073709551616 +Int ( ( ( #getRange ( BM , ADDR , 8
    ) <<Int 56 ) modInt 18446744073709551616 >>Int 56 <<Int 56
     ) modInt 18446744073709551616 +Int 0 ) modInt
    18446744073709551616 ) modInt 18446744073709551616 )
    modInt 18446744073709551616 ) modInt 18446744073709551616
    ) modInt 18446744073709551616 ) modInt
    18446744073709551616 ) modInt 18446744073709551616 )
    modInt 18446744073709551616
```

To make the proof go through, we will need to tell $\mathbb{K}$ how to reduce both the above `i64` expression a bit, and how to reduce the resulting `#set` expressions.

Now, our proof obligation states only that `BM` rewrites to *something*—even if it ends up being a really big expression. See line 26. So even if the memory rewrites

[14] We are currently looking at adding more intermediate categories to the lexicographic order to allow similar shrinking or pushing down of other operators, like multiplications and additions. (These are really general cases of shifts.)

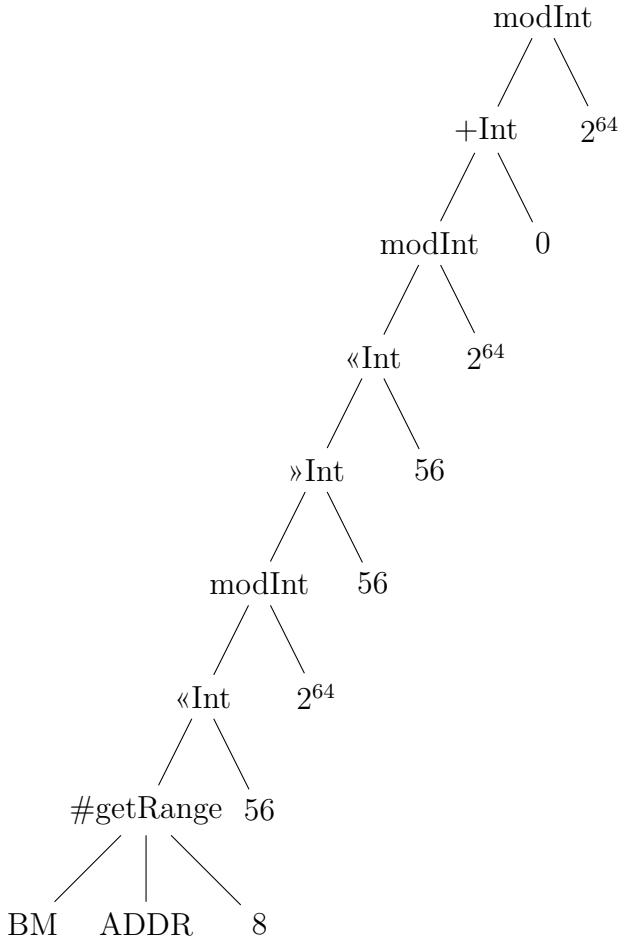


Figure 3.1: The symbolic value of the result of `$i64.reverse_bytes` after one loop iteration. This expression is intended to have shifted the least significant byte of the input, `#getRange(BM, ADDR, 8)`, to be the most significant byte of the result.

to a complex expression, it still satisfies that part of the spec.

The crux comes in the `ensures` section. $\mathbb{K}$ cannot immediately see that the postconditions hold. It is not immediately obvious to that, for example, `#get(BM, ADDR +Int 1)==Int #get(?BM', ADDR +Int 6)`. $\mathbb{K}$ then discharges the proof obligation to Z3 and asks it to prove the postcondition. But since Z3 does not understand our byte map functions, `#get` and `#set`, nor the bit shifts over integers, it does not stand a chance.

So it is time for us to start doing axiom engineering. Our job is to ask ourselves: "What are some true things that the prover does not seem to get?" and then expand its knowledge base while sticking to the 3 principles we set out for ourselves.

Let us look at part of the resulting expression. The algorithm started with `<i64 > 0` in the `local 2`. After one iteration of the loop, the value has changed to the following, representing a single shifted byte:

Listing 3.14: The integer interpretation of a single shifted byte.

```
1 ((((((  #getRange(BM, ADDR, 8) <<Int 56) modInt
     18446744073709551616) >>Int 56) <<Int 56) modInt
     18446744073709551616) +Int 0) modInt 18446744073709551616
```

Here it is as a syntax tree, with 18446744073709551616 converted to 2^{64}:

Looking at this expression, there are some obvious structural changes that we

can tell $\mathbb{K}$ about. We do not include proof of our new axioms here, but there is a formal proof included with the proof in our lemmas file at `https://github.com/kframework/wasm-semantics/blob/master/kwasm-lemmas.md`.

First, let us get rid of the `+Int 0`. Z3, of course, knows that `x + 0` is `x`, but until today, we have not given these reasoning capabilities to $\mathbb{K}$ directly. We are moving to add more reasoning to $\mathbb{K}$ for a variety of reasons. So we open by adding the following, obvious claim to our set of axioms.

Listing 3.15: New axiom: addition by zero.

```
1 rule X +Int 0 => X
```

Then, we have a `modInt` outside a `modInt`. They are even modulus the same number. We could say `(X modInt N)modInt N => X modInt N`, but let us be a bit more general:

Listing 3.16: New axiom: seqences of `mod` operations.

```
1 rule (X modInt M) modInt N => X modInt M
2   requires M >Int 0
3    andBool M <=Int N
```

We also have a left-shift followed by a modulus. In unbounded integers, shifting left by `N` is the same as multiplying by `2^N`. That gives us the following rule:

Listing 3.17: New axiom: left shift followed by `mod`.

```
1 rule (X <<Int N) modInt POW => (X modInt (POW /Int (2 ^Int N)
     )) <<Int N
2   requires N  >=Int 0
3    andBool POW >Int 0
4    andBool POW modInt (2 ^Int N) ==Int 0
```

This gives us a much smaller state, seen in Fig. 3.2

This presents a nice opportunity to get rid of some shifts. Again, recall that these are unbounded integers, so shifting left does not get rid of any bits of information.

Listing 3.18: New axioms: rules for shifts.

```
1 rule (X <<Int N) >>Int M => X <<Int (N -Int M)   requires N
     >=Int M
2 rule X <<Int 0 => X
```

The state is further reduced, by deleting two of the shifts. This exposes yet another situation where we have a `(X modInt 2 ^Int 8)modInt 2 ^Int 8` expression, which gets simplified.

We also tell $\mathbb{K}$ something about the way getting values from (little-endian) memory works:

Listing 3.19: New axiom: getting a single byte.

```
1 rule #getRange(BM, ADDR, WIDTH) modInt 256 => #get(BM, ADDR)
2   requires WIDTH =/=Int 0
3    andBool #isByteMap(BM)
4   ensures  0 <=Int #get(BM, ADDR)
```

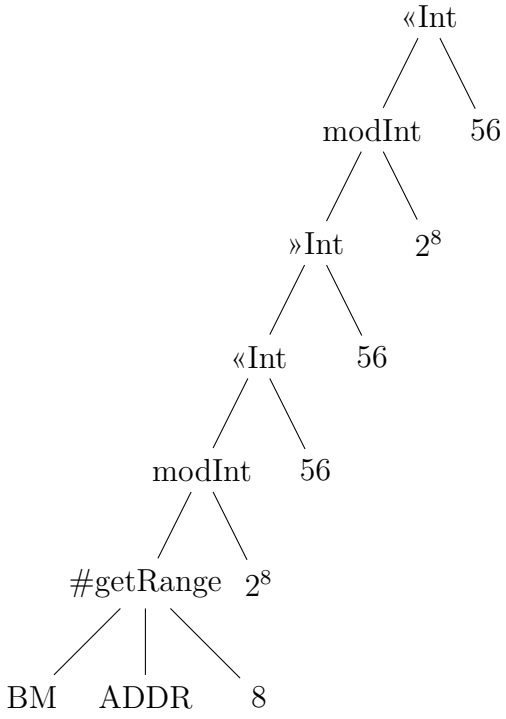


Figure 3.2: The resulting expression after applying the first three simplifications.

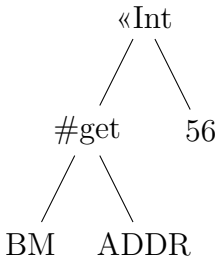


Figure 3.3: The fully simplified expression of the first iteration.

```
5     andBool #get(BM, ADDR) <Int 256
```

In the end, adding these rule leaves us with our final expression (for now):

You may recognize this as getting the least significant byte of the stored value and putting it in the position of the most significant one in the resulting `i64` value. A good start!

3.5.2.3 The Full Set of Extensions

In the end, to get the proof to pass we added 40 new axioms. You can see them all in our lemmas file.

- 25 of these are general lemmas that can be upstreamed into $\mathbb{K}$'s reasoning capabilities.
- 7 relate to the `#get` and `#set` operations of KWasm and can be used in any KWasm verification
- 8 are specific to the proof we just wrote.

Of the 25 general ones, many are just trivial copies of each other. For example, we need both these rules:

```
rule X +Int 0 => X
```

```
rule 0 +Int X => X
```

This repeats for all rules over addition. We cannot tell $\mathbb{K}$ directly that addition commutes, because, if you recall, the rule `X +Int Y => Y +Int X` would cause infinite rewrites. So when it comes to stating true, simple things that $\mathbb{K}$ should know about, we need to be a little repetitive.

The rules specific to this proof are a mixed bag. For example, it turns out to be useful to add the following rule when doing this specific proof:

```
rule X /Int 256 => X >>Int 8
```

We probably do not always want to do this in every semantics, or not even in ever KWasm proof. Therefore, it is made specific to the current proof. Other rules are for the moment very specific, but could possibly be made more general in the future, such as the following rule, which helps get rid of `#get` operations that will be canceled by shift operations.

```
rule (#get(BM, ADDR) +Int X) >>Int 8 => X >>Int 8
  requires X modInt 256 ==Int 0 andBool #isByteMap(BM)
```

We would like to conduct more proofs before deciding which are general enough to warrant being promoted to general KWasm lemmas or even general $\mathbb{K}$ lemmas.

4

Discussion

We were able to almost fully mechanize Wasm in $\mathbb{K}$, except for certain floating-point functionality. This required mapping certain dependencies between different types of declarations. We also managed to mechanize a blockchain embedding and run conformance tests on it. The method of embedding can serve as a template for future language embeddings and requires only a little work for every host function—the same kind of work that one would expect in creating an API for already implemented functionality.

We also managed to use the symbolic execution engine our semantics generated to prove properties of several small programs. One of the programs was a real helper function which did something interesting: it performed byte reversal of integers. This required adding 25 general axioms that are relevant to all $\mathbb{K}$ based proving, 7 axioms that are general to KWasm, and 8 special axioms. The main challenge has turned out to be dealing with linear memory, and most of our proving time and effort has been spent on finding the correct axioms for dealing with the gap between byte-addressable memory and the 32/64 bit words of Wasm.

4.1 How Suitable is KWasm for Verifying Smart Contracts?

This project is missing its proverbial capstone. While we have made progress on making on finishing KWasm, making an Ethereum embedding, and writing proofs for pure Wasm programs, we have not yet verified a smart contract. We must conclude that after this project, KWasm and KEwasm are not yet mature enough to automatically verify the WRC20 contract's main functionality. However, the progress we have made makes us confident that it will be with only a little more development. We hypothesize is that the lemmas on memory we have added will be useful in verifying the argument passing between the Wasm runtime and the EEI, which uses linear memory.

4.1.1 Comparison to KEVM

Proving properties of smart contracts has come a long way in KEVM, unlike in KWasm. There the strength of the automatic process has manifested in the repository of verified smart contracts[1] maintained by Runtime Verification, Inc., where

[1] `https://github.com/runtimeverification/verified-smart-contracts`

many different smart contracts with similar specificatons—such as ERC20 tokens—have been verified for generic properties. With a good set of lemmas in place, it has proven feasible to prove many properties with relatively little extra effort when the property is applied to a new but similar contract.

Another strength of the KEVM is its use of a domain-specific language (DSL) for Ethereum, called EDSL[2], which deals with converting human-readable data structures into EVM bytes. For example, calling a specific function in a smart contract is generally performed by hashing the function signature and taking the first four bytes, called the "function selector", encoding the parameters as bytes, and concatenating the selector and encoded parameters and passing it as call data. Upon execution of the contract, it inspects the first four bytes of the call data and jumps to the corresponding function code. This is a common pattern in EVM contracts, and the DSL has a function for turning a function signature and parameters into call data. When we have used KEwasm for running contracts, for example for making unit tests for the WRC20 contract, we have simply input the data as bytes. Our preliminary experience with trying to prove the main functionality of the WRC20 contract has supported this, in that our specs become verbose due to our lack of useful helper functions for common operations.

4.1.2 Ergonomic Issues of Proving Working with KWasm

We perceived the overall experience of proving with KWasm while adding necessary lemmas as slow due to a lack of interactivity with the tools. One needs to recompile the semantics after adding lemmas and make note of how the states at each step of the prover execution changes with the addition of lemmas. During the course of this project, we were made aware of a Haskell REPL tool for interactive proving, and associated scripts which can unparse the more verbose internal format, called Kore, into more readable $\mathbb{K}$ format. These tools make proving more interactive and gives good overview of the proof process, and we suspect that further development of the REPL and the Haskell backend will improve the ergonomics of proving with $\mathbb{K}$. Indeed, during the course of the project we have seen it mature a great deal, to the point where it is now the main tool we use. We found this very useful for debugging and finding issues with our lemmas, but not yet mature enough to supply an interactive proving experience similar to that of proof assistants. The proof engineer is tasked with setting up initial conditions, running the machine, and inspecting its output, which may take anything from a minute to half an hour to produce. This slowness of iteration currently makes the proof process arduous. It is an issue that is being addressed, but which has been significant during the project.

Another ergonomic issue is that lemmas need to be designed carefully since they always apply. Some more manual control would allow us to, for example, group parentheses left or right as needed for associative operations to make other lemmas apply. Finding a set of lemmas which respect ensures a lexicographic product is always diminishing, as described in Sect. 3.5.2.1, while making the spec pass is time-consuming. Being able to manually apply certain lemmas, such as associativity rules, at given points in the proof would have made matters easier. Again, this is

[2] `https://github.com/kframework/evm-semantics/blob/master/edsl.md`

an issue which is being addressed and we suspect will soon be out of the way.

An ergonomic issue in using the $\mathbb{K}$ prover has been to inspect why a proof fails. Error messages are quite often opaque, and failures to prove a specification has more often resulted in error messages than in the intended, unparsed end states. Using KLab when possible helps a great deal in seeing regular semantic rules apply, but it does not show (or branch on) function application, which is how lemmas are encoded. Therefore, reasoning about why lemmas failed to apply often turned into a game of finding the corresponding Z3 query (printed out when `--debug-z3-queries` is passed to the prover) in the prover output. Since the prover will try to apply all matching functions and emit the side conditions to Z3 this was often feasible, but searching was time-consuming.

In general, the current state of proving in $\mathbb{K}$ requires a good deal of searching through textual output. The transition to using the Haskell backend promises to allow more structured inspection. There is, for example, a REPL for inspecting the symbolic execution of a spec, in which breakpoints can be added and an execution graph can be obtained.

This project has pushed $\mathbb{K}$ into new domains which has led to us running up on the edges of some of the tools. While there is, of course, an EVM semantics and several other high-level language semantics, the combination of byte-based heap reasoning, high-level features, and configuration composition seems to have posed some new challenges. For example, the uncompleted proving efforts focused on using the Ewasm embedding have been stuck on issues with calling out across configurations which we have not yet been able to resolve.

All abstractions are leaky, and $\mathbb{K}$ is no exception. While we would like to keep the semantics as similar to the official Wasm semantics as possible, certain rule formulations—like splitting a rule in two with different side conditions rather than using the builtin `#if ... #then ... #else ... #fi` construct lets the prover branch on the rule when it cannot resolve the condition, whereas if gets stuck on the single rule with `#if`. This leads to certain wordiness and duplication. See for example the changes in pull request #243 into KWasm, especially the splitting of the `#checkTypeUse` rules.

Another way design goals interfere with function is the structure of the KWasm modules, where we include syntax in the semantics module so that we may present syntax productions together with their respective rules. When we tried to introduce a specialized string sort for the kinds of strings that can occur in Wasm, the $\mathbb{K}$ parser broke since it interpreted double-quotes in the Wasm specification—for example when declaring `multiplicity="**"` in a configuration—as these specialized strings. We, therefore, needed to create separate syntax modules that imported the semantics modules and declare the sort `WasmString` without any productions in the semantics module.

4.2 Increasing $\mathbb{K}$'s vs. Z3's Reasoning Capabilities

Our first successful attempts at proving the specs that required extra lemmas used the annotation `smt-lemma`, which extends Z3's reasoning capabilities. Our approach was to make Z3 understand our domain well enough that we could leave as much as possible of the reasoning to the SMT solver. After discussing the matter with $\mathbb{K}$ framework maintainers we decided to change approaches and do the opposite. There was a clear interest in making $\mathbb{K}$ more capable in reasoning about arithmetic and leaving less of the work to Z3. Since Z3 is a black box to $\mathbb{K}$ it cannot be used to construct proof objects, which is a long-term goal for the $\mathbb{K}$ prover.

So we changed our efforts to adding lemmas to $\mathbb{K}$. This was extra time-consuming in the last proof, shown in Sect. 3.5, which was the most complicated proof undertaken during the project. Adding as many SMT lemmas as possible took about as many developer days as adding as few as possible.

The result—several useful arithmetic lemmas that we are planning on upstreaming into the $\mathbb{K}$ prover—was worth the while. Not only does it make the $\mathbb{K}$ prover more powerful, but we also conjecture it will make it faster, though this has not yet been benchmarked. The lemmas reduce the state of the proof obligation which means the size of the expression the prover is working on is reduced on all branches.

We took care to construct hand-written proofs of correctness to these lemmas, in place of proving them with the $\mathbb{K}$ prover. See Sect. 4.4.2 for discussion about the future direction in terms of proving lemmas.

4.3 Issues With the Core Test Suite

The core test suite supplied by the Wasm project has certain limitations. It is based on the reference interpreter and makes unnecessary assumptions about the features of an implementation. For example, tests of parsability, validity, and linkability are mixed with tests of correct execution. For modularity purposes, we advocate that the tests are split up as parser tests, validity tests, linkability tests, and execution tests. KWasm is a tool that assumes validity, and a full-fledged Wasm implementation using KWasm would likely use a separate validity checker. Furthermore, tests should be separated into pure Wasm tests, and tests using host function calls. The issue with mixing these types of tests is that it assumes that a specific host embedding is available, with specific and—as far as we can tell—undocumented host functions. Finally, the tests could also be split up over more files, which would facilitate running them with a simpler setup. It also avoids possible issues, such as insufficient memory, that large files can cause even in valid Wasm implementations—implementations may be limited in many different ways[31, Sect. 7.2].

Furthermore, we would like to see a specification of the host modules that get called in the core test suite, ideally by giving them an operational semantics. This is a good use case for executable semantics, such as those written in $\mathbb{K}$.

4.4 Future work

This project is has been about taking an existing prototype and bringing it off the ground, making it capable of doing useful work. Some work remains to fully complete the current project, namely verifying the WRC20 contract. Other than that, the completed Wasm semantics and Ewasm embedding leave us with many interesting directions to take in the future.

4.4.1 WRC20 Contract

Completing the capstone of this project is the next priority. We have made preliminary attempts at verifying the other WRC20 functions apart from the `$i64.reverse_bytes`. The prover is not yet able to symbolically execute the code of the functions, but halts at an earlier state when declaring the imported Ethereum functions.

The current plan is to write a proof for a smaller program that interacts with the Ethereum environment in some way, and incrementally add proofs that manipulate call-data, that uses a selector, or returns values. KWasm is the first $\mathbb{K}$ semantics of a language that is be used embedded in other semantics. Any proofs of embedded Wasm using KWasm would be a first and will serve as a prototype for future embeddings of KWasm.

4.4.2 Proving Our Lemmas Sound

The added lemmas have handwritten proofs adjacent to them in the source code. Here is an example of a lemma together with its proof, from the `kwasm-lemmas.md`[3] file:

Listing 4.1: A lemma with its handwritten proof

````
1  ```k
2      rule (X modInt M) modInt N => X modInt N
3        requires M >Int 0
4         andBool N >Int 0
5         andBool M modInt N ==Int 0
6        [simplification]
7  ```
8
9  Proof:
10
11 ```
12 Assume m = n * k for some k > 0.
13 x = m * q + r, for a unique q and r s.t. 0 <= r < m
14 (x mod m) mod n
15  = r mod n
16  = (n * ( k * q) + r) mod n
````

[3] https://github.com/kframework/wasm-semantics/blob/342f86d08ef9d9310d4ba66f963864b63fdff105/kwasm-lemmas.md

```
17  = m * q + r mod n
18  = x mod n
19  ```
```

This uses standard facts of algebra, the behavior of the `modInt` operation (selecting the unique positive smallest remainder) and the side conditions to prove the rewrite is sound. This may be fragile. It is not guaranteed, for example, that the backend implements the `mod` function in this way. It may, for example, select a negative remainder. It is also too easy to mix up algebraic properties of congruence classes with the computational behavior of a `mod` function. Nested `mod` operations do not necessarily even make sense in standard arithmetic over congruence classes, where all operands are assumed to be congruence classes over a single positive integer.[12, Sect. 11]

We will want to prove our lemmas—at the moment, they are really axioms of our theory. Ideally, this would be an automatic or semi-automatic process. One published approach manages to automatically turn "tactlets" into proof obligations, which are then validated against a rewriting semantics in Maude[3]. The lemmas that use the KWasm operational semantics—currently, that would be those symbolically unravels memory accesses—could be amenable to a similar approach, where we could show that any application of the lemmas would render an identical result as a sequence of rules from the semantics. There are ongoing efforts to merge several operational semantic rules in KWasm into tactlet-like rules for verification purposes[4]. This approach may still be overkill since all our lemmas are pure functions rather than operational semantic rules. The simplest approach would be to translate the arithmetic statements into SMT formulas and have Z3 validate them.

4.4.3 Upstreaming Lemmas

KWasm is currently the only project adding lemmas for upstreaming into $\mathbb{K}$. The lemmas we have added serve our purposes well and have strong arguments of soundness attached. Still, there may be objections to using some of our lemmas due to different design choices in different languages. For example, `X >>Int N` could be better represented as `X /Int (2 ^Int N)` in some languages, while others may the other way around. The user base of $\mathbb{K}$ is still relatively small, and it would be feasible to make a tentative committee decision on what principles should apply for adding axioms.

4.4.4 Interactive Proving

A major inconvenience in this verification effort is that the $\mathbb{K}$ prover is non-interactive. The lexicographic product in Sect. 3.5.2.1 is a complicated solution to avoid infinite rewrites. If we could gain some control of when certain lemmas apply, or rather when they do not apply, we could do away with the constraints of the lexicographic product and instead use manual verification, supplying a list of do-not-apply instructions with our proof to make the prover go through. Assuming the lemmas are sound,

[4] `https://github.com/runtimeverification/polkadot-verification/issues/42`

applying them can not change the meaning of the program, and the only possible difference between applying and not applying certain lemmas would be whether the prover manages to decide if the spec is correct, not the result of that decision (which is either `#True` or `#False`).

One possible solution would be to have the prover output a table of which lemmas it applied in which steps, and to what expressions, until execution halts. This table would be human-readable and editable, and the verifier could remove entries. By supplying an edited table, the prover would only apply the table-specified lemmas in each step.[5] With such a feature in place, the verifier could prototype the different lemmas and add lemmas that would otherwise cause infinite rewrites. In a certain step, for example, it may be convenient to right-associate additions, commute a multiplication, or similar. If the prover could also be halted at different nodes in the execution tree, the set of lemmas for that set changed, and the prover re-run, it would be a significant step towards an interactive prover.

4.4.5 An Ewasm DSL and Spec Language

In Sect. 4.1.1 we discussed the DSL used in KEVM for simplifying spec writing, called EDSL. We plan on making a similar DSL for KEwasm. The initial efforts of proving embedded KWasm have become verbose and clunky. We would prefer a more human-readable language.

For EVM contracts, there exist many different spec languages that differ in expressive power and ergonomics. KLab uses the ACT[6] language, which translates into KEVM specs, while the verified smart contracts in the Runtime Verification repository uses `.ini` macro files[7] for the same purposes. VerX[30] uses a temporal logic language in the style of JavaScript with function-style combinators. We would like to survey the field of specification languages to decide on a suitable spec language for KWasm or build our own.

4.4.6 Benchmarking Lemmas

We would like to test our hypothesis that simplifying lemmas (which strictly reduce expression size) makes the prover faster or less resource-hungry. We could do this by setting up a series of verification claims which use modular arithmetic that can pass by leaving all the arithmetic reasoning to Z3, and applying early simplifications. We could also simulate the same situation by setting up different claims with different sized expressions and evaluate whether the time spent proving them goes down. Such an analysis should split up the benchmarks between time spent in symbolic execution and time spent in Z3. Yet another possible benchmarking method would be to use the proof of `$i64.reverse_bytes` and switch to using Z3 lemmas instead of $\mathbb{K}$ lemmas, and measure how proving time is affected, and how much time is

[5] The question arises what to do if the verifier modifies the table so that it contains a lemma that does not apply. Initially, the prover may be naive and simply fail in such cases, or report an error and move on. The important first step is to get *some* control over the prover, not building a sophisticated interactive prover.

[6] `https://github.com/dapphub/klab/blob/master/acts.md`

[7] `https://github.com/runtimeverification/verified-smart-contracts`

spent on symbolic execution compared to querying in Z3. If our hypothesis that applying simplification lemmas tend to shorten execution time hold, then we should see time spent on symbolic execution going up, even though the number of lemma applications goes down.

4.4.7 JavaScript Embedding

While we have focused on smart contract verification in this project, Wasm is not primarily known as a smart contract language. It is made to a portable bytecode for many different platforms. As its name suggests, its most prominent use case is for near-native speed computations on the web, embedded in JavaScript[17]. To the best of our knowledge, there is little interest in formally verifying specific Wasm programs on the web, and for use cases such as cryptography, there are other approaches to ensure certain properties, for example, type-based approaches to guaranteeing constant-time computation to avoid side-channel attacks[39]. Nevertheless, with both a Wasm and JavaScript semantics in place in Wasm, we conjecture it would require few development hours to create an embedding semantics, should the need arise, or as a proof of concept of the composability of $\mathbb{K}$ semantics for real-world languages.

5
Conclusion

This project has been guided by the overall target of verifying an Ewasm token smart contract, WRC20. While that goal has not yet been met, several strides have been made towards it.

- KWasm has been completed, giving an almost full Wasm semantics, omitting only certain floating point functionality.
- The semantics have been tested against the conformance test suite, giving unclear results—certain fixes have been made, but a full test run would require implementing a specific embedder or hooking KWasm up to a JavaScript engine.
- An Ethereum embedding for KWasm, KEwasm, that gives meaning to the WRC20 code, has been constructed and tested.
- Several pure Wasm programs have been fully verified, including sequences of memory manipulations.
- Foremost of the verified programs is a single helper function from WRC20.

5.1 Contributions to Wasm and Ethereum

We believe that the following contributions stand out as helpful to the future of verifying Wasm and Ewasm:

5.1.1 Examples of Verifying in Wasm

As mentioned in Sect. 1.4, we are not aware of any other framework for verifying Wasm programs. With this project, we have verified several small programs using demonstrably sound axioms and a well-known verification framework. While there must be little doubt that such verification can be done—Wasm has a formal semantics, after all—engineering examples are nevertheless useful. The official specification of Wasm uses a similar memory representation as KWasm and storing to and loading from memory uses similar recursive functions to `#setRange`. Thus, any Wasm verification manipulating memory would need similar reasoning capabilities to those of $\mathbb{K}$, after our extensions. Our collection of lemmas can, therefore, serve as a basis for other verification projects.

5.1.2 Ewasm Embedding

The Ewasm embedding of KWasm not only serves as a prototype virtual machine but also as a formal semantics of Ewasm. We have given a rewriting based operational semantics of the interaction of the EEI and Wasm, a similar formalism

to the one officially used by Wasm. The Ethereum community could accept the Ewasm formalization as canonical and have a runnable, readable formal specification. Furthermore, it offers an example of how embedder for Wasm can be written in general using rewriting rules. The current JavaScript embedding specification[26] uses JavaScript syntax to specify behavior. When building a verification engine it may be useful to use the same formalism all the way. For example, our embedding approach with synchronizations could be used to obtain a prover for JavaScript with embedded Wasm by combining KJS [28] with Kwasm.

Our approach uses synchronization mechanisms to avoid verifying many possible interleavings. If parallel execution is to be allowed between the embedder and the Wasm execution, some care must be taken that reads and writes to memory, globals and tables are synchronized. These are readable and writable from the embedder, and memory and globals are writable from inside Wasm as well, while tables are simply readable.

5.1.3 Evaluation of the Wasm Tests as Conformance Tests

We found in our evaluation of the conformance tests in Sect. 2.2.1 that the two host modules `test` and `spectest` are undocumented, yet required to pass the test suite. We found that for testing our semantics it would be beneficial if the tests were split up more granularly, with independent modules in different files.

5.2 Contributions to $\mathbb{K}$ and KWasm

Zooming in, we have made the following contributions to the tools we have used for the project: the $\mathbb{K}$ framework in general and to KWasm in particular.

5.2.1 Completing KWasm

This project finalized KWasm into a ready state by adding module support. The only missing features of KWasm are floating-point operations, which we do not intend to support unless a clear use case materializes. For now, the main use case is verifying smart contract and blockchain client code, which makes no use of floating-point operations.

5.2.2 Lemmas for Integer and Modular Arithmetic

We contributed 25 lemmas to KWasm which can be safely upstreamed into $\mathbb{K}$ as general axioms for integer and modular arithmetic. These mostly act as pure simplification, reducing the size of expressions and therefore likely also execution time (though this hypothesis is still untested, see Sect. 4.4.6). Which lemmas should be upstreamed into $\mathbb{K}$ and how to evaluate that this causes no breaking changes are still being discussed.

We have added seven other lemmas local to KWasm which deal with loading from and storing to memory. These have proven useful to verify two programs that modify memory, and we believe they will be useful for future such programs as well. Since

memory is where heap objects live in many Wasm programs, especially complied ones, we believe these lemmas will prove useful in future verification efforts.

We also developed a lexicographic order which shows that our lemmas do not cause infinite rewrites. While this order may be changed in the future depending on the needs of $\mathbb{K}$ we believe this puts us on a solid foundation where we can discuss the merits and pitfalls of different standard lemma sets, and think formally about what changes may cause the prover to loop forever.

5.2.3 Embedded $\mathbb{K}$ Semantics

KEwasm is the first example of taking to real-world semantics—an Ethereum client and Wasm—and combining them into a useful semantics. As such, it works as an example of how $\mathbb{K}$ can allow for this, as well as a spearhead for proofing and benchmarking the semantics combining features.

Bibliography

[1] Contract ABI specification — Solidity 0.5.3 documentation, Oct 2019. [Online; accessed 21. Mar. 2020].

[2] W. Ahrendt, G. J. Pace, and G. Schneider. Smart contracts: A killer application for deductive source code verification. In *Principled Software Development*, pages 1–18. Springer, 2018.

[3] W. Ahrendt, A. Roth, and R. Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 412–426. Springer, 2005.

[4] A. M. Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies*. O'Reilly Media, Inc., 2 edition, 2017.

[5] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on Ethereum smart contracts. *IACR Cryptology ePrint Archive*, 2016:1007, 2016.

[6] B. Beckert and R. Hähnle. Reasoning and verification: State of the art and current trends. *IEEE Intelligent Systems*, 29(1):20–29, 2014.

[7] D. Bogdanas and G. Roşu. K-Java: a complete semantics of Java. *ACM SIGPLAN Notices*, 50(1):445–456, 2015.

[8] T. Bray, J. Paoli, C. Sperberg-McQueen, Y. Mailer, and F. Yergeau. Extensible markup language (XML) 1.0 5th edition, W3C recommendation, November 2008.

[9] I. S. Committee et al. 754-2008 IEEE standard for floating-point arithmetic. *IEEE Computer Society Std*, 2008:517, 2008.

[10] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[11] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[12] J. R. Durbin. *Modern algebra: An introduction*. John Wiley & Sons, 2005.

[13] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *ACM SIGPLAN Notices*, volume 47, pages 533–544. ACM, 2012.

[14] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 328–343. ACM, 2017.

[15] G. Gentzen. Untersuchungen über das logische Schließen. i. *Mathematische zeitschrift*, 39(1):176–210, 1935.

[16] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*, volume 7. Cambridge university press Cambridge, 1989.

[17] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM.

[18] C. Hathhorn, C. Ellison, and G. Roşu. Defining the undefinedness of C. In *ACM SIGPLAN Notices*, volume 50, pages 336–345. ACM, 2015.

[19] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, et al. KEVM: A complete formal semantics of the Ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[21] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[22] A. Israeli and D. G. Feitelson. The Linux kernel as a case study in software evolution. *Journal of Systems and Software*, 83(3):485–501, 2010.

[23] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun. Executable operational semantics of Solidity. *CoRR*, 2018.

[24] N. Martı-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[25] B. McFadden, T. Lukasiewicz, J. Dileo, and J. Engler. Security Chasms of WASM. Technical report, 08 2018. [Online; accessed 30. Jul. 2019].

[26] Ms2ger. WebAssembly JavaScript interface. Technical report, Jan 2020. [Online; accessed 29. Jan. 2020].

[27] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[28] D. Park, A. Stefănescu, and G. Roşu. KJS: A complete formal semantics of JavaScript. In *ACM SIGPLAN Notices*, volume 50, pages 346–356. ACM, 2015.

[29] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu. A formal verification tool for Ethereum VM bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 912–915. ACM, 2018.

[30] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. VerX: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP*, pages 18–20, 2020.

[31] A. Rossberg. WebAssembly specification. Technical report, 2019. [Online; accessed 21. Aug. 2019].

[32] G. Roşu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In M. Johnson and D. Pavlovic, editors, *Algebraic Methodology and Software Technology*, pages 142–162, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[33] G. Roșu and T. F. Șerbănuță. An overview of the K semantic framework. *The*

Journal of Logic and Algebraic Programming, 79(6):397 – 434, 2010. Membrane computing and programming.

[34] R. S. Scowen. Generic base standards. In *Proceedings 1993 Software Engineering Standards Symposium*, pages 25–34. IEEE, 1993.

[35] A. Stefanescu, S. Ciobaca, R. Mereuta, B. Moore, T. F. Serbanuta, and G. Rosu. All-path reachability logic. volume 15. Episciences. org, 2019.

[36] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu. Semantics-based program verifiers for all languages. *ACM SIGPLAN Notices*, 51(10):74–91, 2016.

[37] M. Swan. *Blockchain: Blueprint for a new economy*. O'Reilly Media, Inc., 2015.

[38] C. Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 53–65. ACM, 2018.

[39] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan. CT-Wasm: Type-driven secure cryptography for the Web ecosystem. *CoRR*, 2018.

[40] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014. [Online; accessed 1. Aug. 2019].

[41] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

A

Reverse Bytes: Final Expression

The following is the final map data produced by the `$i64.reverse_bytes` function on a symbolic input map `BM` and integer address `ADDR`, when reversing 8 bytes of memory.

```
#set(
  #set(
    #set(
      #set(
        #set(
          #set(
            #set(
              #set(BM,
              ADDR, _modInt_(_modInt_((_modInt_(_<<Int_(_>>
                 Int_(_modInt_(_<<Int_(#getRange(BM, ADDR, 8)
                 , 0), pow64), 56), 0), pow64) + _modInt_((
                 _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
                 getRange(BM, ADDR, 8), 8), pow64) , 56), 8),
                  pow64) + _modInt_((_modInt_(_<<Int_(_>>Int_
                 (_modInt_(_<<Int_(#getRange(BM, ADDR, 8),
                 16), pow64), 56), 16), pow64) + _modInt_((
                 _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
                 getRange(BM, ADDR, 8), 24), pow64), 56), 24)
                 , pow64) + _modInt_((_modInt_(_<<Int _(_>>
                 Int_(_modInt_(_<<Int_(#getRange(BM, ADDR, 8)
                 , 32), pow64), 56), 32), pow64) + _modInt_((
                 _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
                 getRange(BM, ADDR, 8), 40), pow64), 56), 40)
                 , pow64) + _modInt_((_modInt_(_<<Int_(_>>
                 Int_(_modInt_(_<<Int_(#getRange(BM, ADDR, 8)
                 , 48), pow64), 56), 48), pow64) + _modInt_((
                 _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
                 getRange(BM, ADDR, 8), 56), pow64), 56), 56)
                 , pow64) + 0), pow64)), pow64)), pow64)),
                 pow64)), pow64)), pow64)), pow64)), pow64),
                 256)),
              (ADDR + 1), _modInt_((_modInt _((_modInt_(_<<Int_
                 (_>>Int_(_modInt_(_<<Int_(#getRange(BM, ADDR,
                 8), 0), pow64), 56), 0), pow64) + _modInt_((
                 _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
                 getRange(BM, ADDR, 8), 8), pow64), 56), 8),
```

```
                pow64) + _modInt_((_modInt_(_<<Int_(_>>Int_(
                _modInt_(_<<Int_(#get Range(BM, ADDR, 8), 16),
                 pow64), 56), 16), pow64) + _modInt_((_modInt_
                (_<<Int_(_>>Int_(_modInt_(_<<Int_(#getRange(BM
                , ADDR, 8), 24), pow64), 56), 24), pow64) +
                _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_
                <<Int_(#getRange(BM, ADDR, 8), 32), pow64),
                56), 32), pow64) + _modInt_((_modInt_(_<<Int_(
                _>>Int_(_modInt_(_<<Int_(#getRange(BM, ADDR,
                8), 40), pow64), 56), 40), pow64) + _modInt_((
                _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
                getRange(BM, ADDR, 8), 48), pow64), 56), 48),
                pow64) + _modInt_((_modInt_(_<<Int_(_>>Int_(_
                modInt_(_<<Int_(#getRange(BM, ADDR, 8), 56),
                pow64), 56), 56), pow64) + 0), pow64)), pow64)
                ), pow64)), pow64)), pow64)), pow64)), pow64))
                , pow64) / 256), 256)),
11          ((ADDR + 1) + 1), _modInt_(((_modInt_((_modInt_(_<<
              Int_(_>>Int_(_modInt_(_<<Int_(#getRange(BM, ADDR
              , 8), 0), pow64), 56), 0), pow64) + _modInt_((
              _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
              getRange(BM, ADDR, 8), 8), pow64), 56), 8),
              pow64) + _modInt_((_modInt_(_<<Int_(_>>Int_(
              _modInt_(_<<Int_(#getRange(BM, ADDR, 8), 16),
              pow64), 56), 16), pow64) + _modInt_((_modInt_(_
              <<Int_(_>>Int_(_modInt_(_<<Int_(#getRange(BM,
              ADDR, 8), 24), pow64), 56), 24), pow64) +
              _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<
              Int_(#getRange(BM, ADDR, 8), 32), pow64), 56),
              32), pow64) + _modInt_((_modInt_(_<<Int_(_>>Int_
              (_modInt_(_ <<Int_(#getRange(BM, ADDR, 8), 40),
              pow64), 56), 40), pow64) + _modInt_((_modInt_(_
              <<Int_(_>>Int_(_modInt_(_<<Int_(#getRange(BM,
              ADDR, 8), 48), pow64), 56), 48), pow64) +
              _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<
              Int_(#getRange(BM, ADDR, 8), 56), pow64), 56),
              56), pow64) + 0), pow64)), pow64)), pow64)),
              pow64)), pow64)), pow64)), pow64)), pow64) /
              256) / 256), 256)),
12          (((ADDR + 1) + 1) + 1), _modInt_((((_modInt_((
              _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
              getRange(BM, ADDR, 8), 0), pow64), 56), 0), pow64)
               + _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<
              Int_(#getRange(BM, ADDR, 8), 8), pow64), 56), 8),
              pow64) + _modInt_((_modInt_(_<<Int_(_>>Int_(
              _modInt_(_<<Int_(#getRange(BM, ADDR, 8), 16),
              pow64), 56), 16), pow64) + _modInt_((_modInt_(_<<
              Int_(_>>Int_(_modInt_(_<< Int_(#getRange(BM, ADDR,
               8), 24), pow64), 56), 24), pow64) + _modInt_((
```

```
          _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
          getRange(BM, ADDR, 8), 32), pow64), 56), 32),
          pow64) + _modInt_((_modInt_(_<<Int_(_>>Int_(
          _modInt_(_<<Int_(#getRange(BM, ADDR, 8), 40),
          pow64), 56), 40), pow64) + _modInt_((_modInt_(_<<
          Int_(_>>Int_(_modInt_(_<<Int_(#getRange(BM, ADDR,
          8), 48), pow64), 56), 48), pow64) + _modInt_((
          _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
          getRange(BM, ADDR, 8), 56), pow64), 56), 56),
          pow64) + 0), pow64)), pow64)), pow64)), pow64)),
          pow64)), pow64)), pow64)), pow64) / 256) / 256) /
          256), 256)),
13      ((((ADDR + 1) + 1) + 1) + 1), _modInt_(((((_modInt_((
          _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#getRange(
          BM, ADDR, 8), 0), pow64), 56), 0), pow64) + _modInt_
          ((_modInt_(_<<Int_(_>>I nt_(_modInt_(_<<Int_(#
          getRange(BM, ADDR, 8), 8), pow64), 56), 8), pow64) +
           _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_
          (#getRange(BM, ADDR, 8), 16), pow64), 56), 16),
          pow64) + _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_
          (_<<Int_(#getRange(BM, ADDR, 8), 24), pow64), 56),
          24), pow64) + _modInt_((_modInt_(_<<Int_(_>>Int_(
          _modInt_(_<<Int_(#getRange(BM, ADDR, 8), 32), pow64)
          , 56), 32), pow64) + _modInt_((_modInt_(_<<Int_(_>>
          Int_(_modInt_(_<<Int_(#getRange(BM, ADDR, 8), 40),
          pow64), 56), 40), pow64) + _modInt_((_mo dInt_(_<<
          Int_(_>>Int_(_modInt_(_<<Int_(#getRange(BM, ADDR, 8)
          , 48), pow64), 56), 48), pow64) + _modInt_((_modInt_
          (_<<Int_(_>>Int_(_modInt_(_<<Int_(#getRange(BM, ADDR
          , 8), 56), pow64), 56), 56), pow64) + 0), pow64)),
          pow64)), pow64)), pow64)), pow64)), pow64)), pow64))
          , pow64) / 256) / 256) / 256) / 256), 256)),
14    (((((ADDR + 1) + 1) + 1) + 1) + 1), _modInt_((((((
          _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
          getRange(BM, ADDR, 8), 0), pow64), 56), 0), pow64) +
          _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
          getRange(BM, ADDR, 8), 8), pow64), 56), 8), pow64) +
          _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
          getRange(BM, ADDR, 8), 16), pow64), 56), 16), pow64) +
           _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
          getRange(BM, ADDR, 8), 24), pow64), 56), 24), pow64) +
           _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
          getRange(BM, ADDR, 8), 32), pow64), 56), 32), pow64) +
           _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
          getRange(BM, ADDR, 8), 40), pow64), 56), 40), pow64) +
           _modInt_((_modInt_(_<<Int_(_>>Int_(_ modInt_(_<<Int_
          (#getRange(BM, ADDR, 8), 48), pow64), 56), 48), pow64)
           + _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_
          (#getRange(BM, ADDR, 8), 56), pow64), 56), 56), pow64)
```

```
        + 0), pow64)), pow64)), pow64)), pow64)), pow64)), pow64)),
       pow64)), pow64)), pow64) / 256) / 256) / 256) / 256) /
        256), 256)),
15  ((((((ADDR + 1) + 1) + 1) + 1) + 1) + 1), _modInt_(((((((
     _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
     getRange(BM, ADDR, 8), 0), pow64), 56), 0), pow64) +
     _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
     getRange(BM, ADDR, 8), 8), pow64), 56), 8), pow64) +
     _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
     getRange(BM, ADDR, 8), 16), pow64), 56), 16), pow64) +
     _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
     getRange(BM, ADDR, 8), 24), pow64), 56), 24), pow64) +
     _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
     getRange(BM, ADDR, 8), 32), pow64), 56), 32), pow64) +
     _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
     getRange(BM, ADDR, 8), 40), pow64), 56), 40), pow64) +
     _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_ <<Int_(#
     getRange(BM, ADDR, 8), 48), pow64), 56), 48), pow64) +
     _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#
     getRange(BM, ADDR, 8), 56), pow64), 56), 56), pow64) +
     0), pow64)), pow64)), pow64)), pow64)), pow64)), pow64))
     , pow64)), pow64) / 256) / 256) / 256) / 256) / 256) /
     256), 256)),
16 (((((((ADDR + 1) + 1) + 1) + 1) + 1) + 1) + 1), _modInt_
    (((((((_modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<
    Int_(#getRange(BM, ADDR, 8), 0), pow64), 56), 0), pow64) +
     _modInt_((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#getR
     ange(BM, ADDR, 8), 8), pow64), 56), 8), pow64) + _modInt_
    ((_modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#getRange(BM,
    ADDR, 8), 16), pow64), 56), 16), pow64) + _modInt_((
    _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#getRange(BM,
    ADDR, 8), 24), pow64), 56), 24), pow64) + _modInt_((
    _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#getRange(BM,
    ADDR, 8), 32), pow64), 56), 32), pow64) + _modInt_((
    _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#getRange(BM,
    ADDR, 8), 40), pow64), 56), 40), pow64) + _modInt_((
    _modInt_(_<<Int_(_>>Int_(_ modInt_(_<<Int_(#getRange(BM,
    ADDR, 8), 48), pow64), 56), 48), pow64) + _modInt_((
    _modInt_(_<<Int_(_>>Int_(_modInt_(_<<Int_(#getRange(BM,
    ADDR, 8), 56), pow64), 56), 56), pow64) + 0), pow64)),
    pow64)), pow64)), pow64)), pow64)), pow64)), pow64)),
    pow64) / 256) / 256) / 256) / 256) / 256) / 256) / 256),
    256))
```

B

The WRC20 Wasm Module

Printed here for completeness is the full WRC20 code, parsed from the concrete syntax into KWasm's internal syntax.

```
( module

  // Imports from host module.

  (func String2Identifier("$revert")
    ( import #unparseWasmString("\"ethereum\"")
      #unparseWasmString("\"revert\"")
    )
    param i32 i32 .ValTypes .TypeDecls
  )

  (func String2Identifier("$finish")
    ( import #unparseWasmString("\"ethereum\"")
      #unparseWasmString("\"finish\"")
    )
    param i32 i32 .ValTypes .TypeDecls
  )

  (func String2Identifier("$getCallDataSize")
    ( import #unparseWasmString("\"ethereum\"")
      #unparseWasmString("\"getCallDataSize\"")
    )
    result i32 .ValTypes .TypeDecls
  )

  (func String2Identifier("$callDataCopy")
    ( import #unparseWasmString("\"ethereum\"")
      #unparseWasmString("\"callDataCopy\"")
    )
    param i32 i32 i32 .ValTypes .TypeDecls
  )

  (func String2Identifier("$storageLoad")
    ( import #unparseWasmString("\"ethereum\"")
      #unparseWasmString("\"storageLoad\"")
    )
    param i32 i32 .ValTypes .TypeDecls
```

```
38  )
39
40  (func String2Identifier("$storageStore")
41    ( import #unparseWasmString("\"ethereum\"")
42      #unparseWasmString("\"storageStore\"")
43    )
44    param i32 i32 .ValTypes .TypeDecls
45  )
46
47  (func String2Identifier("$getCaller")
48    ( import #unparseWasmString("\"ethereum\"")
49      #unparseWasmString("\"getCaller\"")
50    )
51    param i32 .ValTypes .TypeDecls
52  )
53
54  // Instantiate a memory.
55
56  ( memory ( export #unparseWasmString("\"memory\"") ) 1 )
57
58  // Entry point to the module---"main" gets called when
       contract gets called.
59
60  (func ( export #unparseWasmString("\"main\"") ) .TypeDecls
       .LocalDecls
61    block .TypeDecls
62      block .TypeDecls
63        call String2Identifier("$getCallDataSize")
64        i32.const 4
65        i32.ge_u
66        br_if 0
67        i32.const 0
68        i32.const 0
69        call String2Identifier("$revert")
70        br 1
71        .EmptyStmts
72      end
73      i32.const 0
74      i32.const 0
75      i32.const 4
76      call String2Identifier("$callDataCopy")
77      block .TypeDecls
78        i32.const 0
79        i32.load
80        i32.const 436376473:Int
81        i32.eq
82        i32.eqz
83        br_if 0
84        call String2Identifier("$do_balance")
```

```
85          br 1
86          .EmptyStmts
87        end
88        block .TypeDecls
89          i32.const 0 i32.load
90          i32.const 3181327709:Int
91          i32.eq
92          i32.eqz
93          br_if 0
94          call String2Identifier("$do_transfer")
95          br 1
96          .EmptyStmts
97        end
98        i32.const 0
99        i32.const 0
100       call String2Identifier("$revert")
101       .EmptyStmts
102     end
103     .EmptyStmts
104   )
105
106   (func String2Identifier("$do_balance") .TypeDecls .
        LocalDecls
107     block .TypeDecls
108       block .TypeDecls
109         call String2Identifier("$getCallDataSize")
110         i32.const 24
111         i32.eq
112         br_if 0
113         i32.const 0
114         i32.const 0
115         call String2Identifier("$revert")
116         br 1
117         .EmptyStmts
118       end
119       i32.const 0
120       i32.const 4
121       i32.const 20
122       call String2Identifier("$callDataCopy")
123       i32.const 0
124       i32.const 32
125       call String2Identifier("$storageLoad")
126       i32.const 32
127       i32.const 32
128       i64.load
129       call String2Identifier("$i64.reverse_bytes")
130       i64.store
131       i32.const 32
132       i32.const 8
```

```
133       call String2Identifier("$finish")
134       .EmptyStmts
135     end
136     .EmptyStmts
137   )
138
139   (func String2Identifier("$do_transfer") .TypeDecls local
         i64 i64 i64 .ValTypes .LocalDecls
140     block .TypeDecls
141       block .TypeDecls
142         call String2Identifier("$getCallDataSize")
143         i32.const 32
144         i32.eq
145         br_if 0
146         i32.const 0
147         i32.const 0
148         call String2Identifier("$revert")
149         br 1
150         .EmptyStmts
151       end
152       i32.const 0
153       call String2Identifier("$getCaller")
154       i32.const 32
155       i32.const 4
156       i32.const 20
157       call String2Identifier("$callDataCopy")
158       i32.const 64
159       i32.const 24
160       i32.const 8
161       call String2Identifier("$callDataCopy")
162       i32.const 64
163       i64.load
164       call String2Identifier("$i64.reverse_bytes")
165       local.set 0
166       i32.const 0
167       i32.const 64
168       call String2Identifier("$storageLoad")
169       i32.const 64
170       i64.load
171       local.set 1
172       i32.const 32
173       i32.const 64
174       call String2Identifier("$storageLoad")
175       i32.const 64
176       i64.load
177       local.set 2
178       block .TypeDecls
179         local.get 0
180         local.get 1
```

```
181         i64.le_u
182         br_if 0
183         i32.const 0
184         i32.const 0
185         call String2Identifier("$revert")
186         br 1
187         .EmptyStmts
188       end
189       local.get 1
190       local.get 0
191       i64.sub
192       local.set 1
193       local.get 2
194       local.get 0
195       i64.add
196       local.set 2
197       i32.const 64
198       local.get 1
199       i64.store
200       i32.const 0
201       i32.const 64
202       call String2Identifier("$storageStore")
203       i32.const 64
204       local.get 2
205       i64.store
206       i32.const 32
207       i32.const 64
208       call String2Identifier("$storageStore")
209       .EmptyStmts
210     end
211     .EmptyStmts
212   )
213
214   (func String2Identifier("$i64.reverse_bytes") param i64 .
        ValTypes result i64 .ValTypes .TypeDecls local i64 i64 .
        ValTypes .LocalDecls
215     block .TypeDecls
216       loop .TypeDecls
217         local.get 1
218         i64.const 8
219         i64.ge_u
220         br_if 1
221         local.get 0
222         i64.const 56
223         local.get 1
224         i64.const 8
225         i64.mul
226         i64.sub
227         i64.shl
```

```
228           i64.const 56
229           i64.shr_u
230           i64.const 56
231           i64.const 8
232           local.get 1
233           i64.mul
234           i64.sub
235           i64.shl
236           local.get 2
237           i64.add
238           local.set 2
239           local.get 1
240           i64.const 1
241           i64.add
242           local.set 1
243           br 0
244           .EmptyStmts
245         end
246         .EmptyStmts
247       end
248       local.get 2
249       .EmptyStmts
250     )
251     .Defns
252 )
```

C

The Complete Configuration of KWasm

Listing C.1: The complete $\mathbb{K}$ configuration of KWasm.

```
configuration
      <wasm>
        <k> $PGM:Stmts </k>
        <valstack> .ValStack </valstack>
        <curFrame>
          <locals>    .Map </locals>
          <localIds>  .Map </localIds>
          <curModIdx> .Int </curModIdx>
          <labelDepth> 0   </labelDepth>
          <labelIds>  .Map </labelIds>
        </curFrame>
        <moduleRegistry> .Map </moduleRegistry>
        <moduleIds> .Map </moduleIds>
        <moduleInstances>
          <moduleInst multiplicity="*" type="Map">
            <modIdx>      0    </modIdx>
            <exports>     .Map </exports>
            <typeIds>     .Map </typeIds>
            <types>       .Map </types>
            <nextTypeIdx> 0    </nextTypeIdx>
            <funcIds>     .Map </funcIds>
            <funcAddrs>   .Map </funcAddrs>
            <nextFuncIdx> 0    </nextFuncIdx>
            <tabIds>      .Map </tabIds>
            <tabAddrs>    .Map </tabAddrs>
            <memIds>      .Map </memIds>
            <memAddrs>    .Map </memAddrs>
            <globIds>     .Map </globIds>
            <globalAddrs> .Map </globalAddrs>
            <nextGlobIdx>   0    </nextGlobIdx>
          </moduleInst>
        </moduleInstances>
        <nextModuleIdx> 0 </nextModuleIdx>
        <mainStore>
          <funcs>
```

```
36              <funcDef multiplicity="*" type="Map">
37                <fAddr>    0               </fAddr>
38                <fCode>    .Instrs:Instrs </fCode>
39                <fType>    .Type           </fType>
40                <fLocal>   .Type           </fLocal>
41                <fModInst> 0               </fModInst>
42              </funcDef>
43            </funcs>
44            <nextFuncAddr> 0 </nextFuncAddr>
45            <tabs>
46              <tabInst multiplicity="*" type="Map">
47                <tAddr> 0    </tAddr>
48                <tmax>  .Int </tmax>
49                <tsize> 0    </tsize>
50                <tdata> .Map </tdata>
51              </tabInst>
52            </tabs>
53            <nextTabAddr> 0 </nextTabAddr>
54            <mems>
55              <memInst multiplicity="*" type="Map">
56                <mAddr> 0                  </mAddr>
57                <mmax>  .Int               </mmax>
58                <msize> 0                  </msize>
59                <mdata> ByteMap <| .Map |> </mdata>
60              </memInst>
61            </mems>
62            <nextMemAddr> 0 </nextMemAddr>
63            <globals>
64              <globalInst multiplicity="*" type="Map">
65                <gAddr>  0         </gAddr>
66                <gValue> undefined </gValue>
67                <gMut>   .Mut      </gMut>
68              </globalInst>
69            </globals>
70            <nextGlobAddr> 0 </nextGlobAddr>
71          </mainStore>
72          <deterministicMemoryGrowth> true </
               deterministicMemoryGrowth>
73          <nextFreshId> 0 </nextFreshId>
74        </wasm>
```

Index