SEMANTICS OF LOW-LEVEL LANGUAGES

BY

ANDREW MIRANTI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Professor Grigore Rosu

ABSTRACT

In this paper we address the motivations, problems and challenges involved in formally specifying low level programming languages. We utilize the K programming language specification framework to build executable models of the languages discussed: x86 and Tezos Michelson. We extend an existing formalization of x86 to include its most common format - executable binaries by implementing an instruction decoder. We start completely from scratch with another: Tezos' Michelson. We produce executable models capable of running programs in each language, with natural extensions towards formal verification tools possible through the K Framework. Finally, we discuss the differences between the two languages which make formalizing the former daunting, and the latter relatively straightforward.

To my parents, for their love and support.

ACKNOWLEDGMENTS

We would like to thank the Runtime Verification team (`https://runtimeverification.com/`), for their contributions, their patience and their support. We would like to thank the other members of the Formal Systems research group (`http://fsl.cs.illinois.edu/index.php/People`) for answering our countless questions.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

There are many approaches to software quality control. The most basic form, practiced by virtually every member of the industry from the largest teams at software giants to the smallest hobbyist programmers is concrete testing - simply running the program on fixed inputs and checking that it outputs the expected result. But, while better than nothing, concrete testing should seem to lack any sort of rigor. For any non-trivial program, the space of possible inputs so dwarfs the amount of available machine and human time to produce and run tests that testing even a tiny fraction of the total number of inputs becomes utterly infeasible. Moreover, for realistic programs whose inputs reasonably contain elements of system state, the complexity of the possible input spaces makes even imagining all of the possible edge cases impractical, let alone attempting to actually test them all.

However, in spite of the impracticality, certain types of software demand an extreme degree of reliability which would seem utterly impractical to provide using normal concrete testing methods. This necessitates a different approach to software verification. In formal methods, one models a program as a mathematical object and them attempts to prove properties about that object with the assistance of a computer. However, in order to model a program, one must have some rigorous way of describing its behavior. This requires a semantic definition of the language in which the program is specified. These models can be purely mathematical, but that approach has drawbacks - without the ability to execute the semantic model of a language, there would seem to be little reason to trust that it matches the common implementations of the language it describes. This need for a model that is both mathematically rigorous and executable gives rise to the concept of an operational semantics. This paper will discuss two operational semantics of different low level languages. These operational semantics are implemented in the K programming language framework.

1.1 BACKGROUND: THE K FRAMEWORK

K[1] is a framework for specifying the operational semantics of a programming language and generating tools from these semantics. This approach has a number of advantages over traditional methods of formal verification. Traditional methods of formal verification require the invention of (typically non-executable) language-specific proof rules in Hoare logic or the like. Furthermore, implementing these proof systems is commonly a completely separate task from the creation of the tooling required to actually use the language - such as an interpreter. This creates an enormous amount of redundant work, and necessitates equivalence proofs

between these different forms of the language.

K allows for the generation of all the tools one would want to use a programming language including a parser, an interpreter, a program verifier, a symbolic execution engine and more from a single definition. Rather than needing to create these tools separately and then prove them equivalent, K generates them in a correct-by-construction manner from a single source. Hence, assuming the correctness of the K tool itself, the generated constructs will be equivalent by definition.

Furthermore, the layer of abstraction created by the K language allows for a multitude of specialized backends to implement the K language with different priorities. At present, the K backend has been implemented in Java, OCaml, LLVM, and Haskell. Presently, LLVM is the preferred backend for conventional concrete execution as in an interpreter, and Haskell is the preferred backend to perform symbolic reasoning. x86 was created before the LLVM and Haskell backends were finished, and thus uses Java for symbolic execution and OCaml for concrete execution. Michelson uses the new backends exclusively.

K has been used to define operational semantics for a wide variety of real world languages, ranging from relatively low level source languages such as C[2] to very high level languages such as Javascript[3]. More recently it has entered the Blockchain sphere, implementing semantics of the EVM [4]. In this paper we use the K Framework to implement semantics of very low level machine languages. Before we discuss these specific semantics, however, we must what a K semantics is and how it works.

1.1.1 A K Semantics

Syntax

A K Semantics consists of several modules which define a language. The first module we will discuss is the syntax module. This module, as its name implies, defines what sort of strings constitute valid programs in this language. For example, the following syntax is an excerpt from the Michelson syntax module:

```
syntax AmountGroup ::= "amount" Int
```

This K instruction defines a nonterminal "AmountGroup" which consists of the literal token "amount" followed by an Int, a nonterminal whose syntax is defined elsewhere in a K builtin. By using many declarations such as this one, a K syntax module specifies a syntax for the programming language in question. Furthermore, syntax definitions may appear in other modules besides the designated syntax module to help with code organization.

For example, the Michelson semantics uses the following production as a data structure to contain information about a Michelson list.

```
syntax MichelsonList ::= #List(List, Type)
```

As this production does not appear in the specific Michelson syntax module, it would not be valid for a program passed to the K interpreter to contain it. However, the K semantics uses productions of this format at runtime. By convention, the '#' symbol is used for internal productions such as this one. Syntax non-terminals also act as sorts, which creates a K type system separate from the specified languages' type system.

Configuration

In K terminology, a configuration is an XML-like data structure which contains the mutable state of a program in the language during execution. At the very least, a K configuration requires a single 'k' cell, which conventionally contains productions describing the next execution steps to run. In most cases, however, a K semantics will additionally include several other cells such as a map from variable names to values specifying the contents of memory as a program runs. Alternatively configuration cells may contain information about global machine state that a program may need to execute. K rewrite rules (described in the next section) may be considered functions mapping one configuration value to the next one, and in this way a program's execution can be modeled as a sequence (or a tree, if the language described is nondeterministic) of valid configurations joined by the rewrite rules which transition from one configuration to the next. As an example, below is an excerpt from the Michelson configuration:

```
configuration <michelsonTop>
              <k> $PGM:Pgm </k>
              <stack> .K </stack>
              <paramtype> #NotSet </paramtype>
              <paramvalue> #NoData </paramvalue>
              ...
            </michelsonTop>
```

Of note here is the $PGM variable. This variable will be substituted for the initial program parsed according to the syntatic productions in the main syntax module. The ':Pgm' token after the variable indicates the expected sort for the parsed program - or, equivalently,

the starting nonterminal for the K parser. Aside from this variable, the remainder of the K starting configuration is typically constant, such as the contents of the 'paramtype' ad 'paramvalue' cells.

1.1.2 Rewrite Rules

Rewrite rules make up the main body of a K semantics. These rules match certain elements of a configuration and transform the configuration into the next state. Rules consist of a left hand side and a right hand side. The left hand side consists of configuration cells containing both terminals and variables, which creates constraints on the structures of configurations that the rule can rewrite. Variables may appear more than once on the left hand side, in which case K will enforce that the variable has consistent values in all locations. Additionally, rules may have side conditions which create more restrictions on when the rule can be applied. Finally, rules can have attributes which control when or how the rule applies. For example, the 'owise' attribute makes a rule apply only if no other non-owise rules can apply.

The right hand side of the rule describes the configuration it creates. It may contain any of the variables on the left hand side, in which case the appropriate value will be substituted in for the variable. It may additionally create any production in scope. The process of choosing a rule to apply based on its left hand side is referred to as matching. The process of changing the configuration based on a rule is referred to as rewriting. An example rewrite rule is included below, taken from the Michelson semantics.

```
rule <k> EDIV A  => #HandleAnnotations(A) ... </k>
     <stack> I1 ~> I2 => Some (Pair (I1 /Int I2) (I1 %Int I2)) ... </stack>
     requires I2 =/=Int 0
```

This rule matches the 'EDIV' literal followed by a variable 'A' of sort 'AnnotationList' (this sort is inferred from the type of the parameter of the '#HandleAnnotations' production). The ... at the end of the 'k' and 'stack' cells signifies that these cells contain a sequence of items, and there may be more items not specified in the rule. Without it, this rule would apply only if the k and stack cells had only the elements specifically matched in the rule. In the stack cell, this rule matches sequences starting with two Ints which are labeled 'I1' and 'I2.' The 'requires' statement at the end specifies that this rule matches only when 'I2' is not zero, which avoids a division by zero case which the Michelson semantics handles separately. The variables A, I1 and I2 are reused on the right hand side of this rule. Note that this rule only matches on the 'k' and 'stack' cells of the configuration. The remaining cells in the configuration are not modified and do not participate in matching, so we are not compelled

to mention them. This feature of K allows us to change the configuration without modifying existing rules most of the time.

CHAPTER 2: TOWARDS A SEMANTICS OF X86 BINARIES

The great prize in the realm of low-level language semantics is almost certainly x86_64. While most programs today are written in far higher-level languages: C, Java, Python and the like, ultimately none of these languages are directly executable. A compiler or interpreter must translate all of them ultimately to a lower level, more physically realizable representation. So, while one can specify and formally verify programs in a higher-level language, one must take any conclusions drawn from such formalizations with a large grain of salt. The translation process from the higher-level representation to the actual execution language is far from trivial, and the possibility of bugs being introduced during this process cannot reasonably be captured in verification over higher level languages. This then implies a need to perform verification directly on lower level languages.

Unfortunately, many of these languages are not well suited to verification. As an example, x86_64 is an old and highly complex language – with thousands of different instructions specified in a wide variety of differing formats. The behavior of these instructions is specified only in semi-formal technical manuals (as opposed to executable semantics), and is frequently operating system, processor model and even processor mode dependent. In a sense, the problem of formalizing x86 is not well defined because x86 cannot reasonably be considered a single language, rather a family of languages sharing a name, developed in parallel by multiple different organizations. Formalization is further frustrated by the errors in these documents and in other representations of the language. Such errors are essentially unavoidable in the absence of executable semantics, as one cannot automatically verify the correctness of a pdf file. With processor errata, documentation errors, and varying implementation details changing from processor to processor, even determining what the ground truth is should seem daunting.

It should then come as no surprise that most existing semantics or formalizations of x86_64 make drastic simplifying assumptions, such as simply omitting large subsets of the instruction set to reduce the scope of the problem. Yet, in doing so, these formalizations greatly limit their own utility – since compilers typically must generate these omitted instructions. While certainly the base set of instructions is far more prevalent in programs than the long tail of varying forms of multimedia and vector extensions added on over the years, these extensions are still used routinely – sometimes not even intentionally on the part of the program creator. Even supposedly obsolete instruction subsets can still be found as compatibility features. This leads to other formalizations taking a different approach – trying to find ways of automatically formalizing large sections of the x86_64 language. We followed

this second approach, extending a previous formalization of x86 by Sandeep Dasgupta with a novel formalization of the x86 decoder based upon Intel's XED decoder [5]. This decoder was produced in cooperation with Sandeep Dasgupta of the original K x86 semantics.

2.1 PRIOR WORK

The instruction decoder described in this paper is implemented as an extension to the x86 execution semantics implemented by Dasgupta et al. [6]. This semantics is in turn builds upon work done by Heule et al. [7] to automatically generate SMT formulae for much of the x86 instruction set automatically.

2.1.1 Strata

The semantics of x86 extended in this paper are based upon the work of Heule et al. [7] to formalize a wide swath of x86 using automatic generation of instruction semantics through the use of automatic program generation.

Their basic approach is to manually specify the semantics for a relatively small number of x86 instructions called the base set. They augment the base set with several pseudo-instructions for manipulating parts of the x86 processor state that the real instruction set does not directly permit one to modify. For each instruction in the target set, they first generate a set of test cases detailing the instruction's behavior on some arbitrarily chosen inputs. They then feed these test cases along with the base instructions to an automatic program synthesizer called Stoke, which attempts to learn a loop-free program whose behavior on the test cases is identical to the single instruction.

They transform this program into an SMT formula and then repeat the process to generate another such formula. They check to see if these formulae are equivalent using an SMT solver and if they are not, they use the counterexample output by the SMT solver as a new test case, rejecting whichever program's output differs from the real instruction on this case (or perhaps both programs, if both differ). In the event that the SMT solver cannot prove the programs equivalent or produce a counterexample, they create equivalence classes of programs and select a formula from one of the equivalence classes depending on a number of factors (number of uninterpreted functions, number of non-linear operations, etc) which make certain formula more or less difficult to reason with using an SMT solver.

Finally, after they have learned a formula for a target instruction, they add it to the base set and attempt to learn more instructions using the newly specified one. This gives rise to the term "Stratified Synthesis." They then perform a generalization step that attempts to

implement other variants of the same instruction using one learned variant. Overall, they automatically produce formulae for nearly 1,800 instruction variants using an initial input set of only 62 total instructions and pseudo-instructions.

2.1.2 A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture

This execution semantics of x86_64 in the K framework by Dasgupta et al. [6] forms the basis of our instruction decoder. Among the most comprehensive semantics of the x86_64 language, it attempts to formalize nearly the entire user-level instruction set of the language. It covers a total of 3155 instruction variants, with 774 unique assembler mnemonics. Notable exceptions include:

1. Some deprecated instructions (e.g. x87, MMX) due to considering them an unacceptable amount of work when compared to the number of useful programs that would still utilize them.

2. System level instructions in general, but additionally the syscall instruction due to the lack of any formal specification for the operating system

3. Concurrency related instructions (in particular, atomic operations and the LOCK prefix) due to the absence of any representation for parallel processing in the semantics.

4. Some special purpose instructions such as hardware cryptography.

It builds upon the original Strata semantics, translating them into K and furthermore simplifying and generalizing Strata's automatically generated rules. In particular, they combined the Strata rules for many of the operations with 8-bit immediate instructions, which Strata implements by creating a separate formula for each possible value of the immediate argument. Furthermore, they extend the Strata semantics with both automatically generated and manually written semantic rules. As a K semantics, the x86 instructions are formalized as rewrite rules over a configuration designed to mimic the internal state of an idealized x86 processor.

They then validated these new semantics rules by using K to translate the rewrite rules back to SMT formulae, and then performing SMT equivalence checking against the original Strata formulae. They furthermore tested each instruction on several thousand test inputs and compared the results to hardware. Finally, they verified the semantics by executing the GCC C-torture tests (excluding a few which use systems level instructions). They needed to re-implement some C library functions either natively in K or in a miniaturized C standard

library in order to execute all of these tests. In performing this validation, they found a number of flaws in the Intel Manual and other x86 semantics. In particular, they found flaws related to incorrect generalizations between different variants of the same instruction.

They discuss a number of possible applications for the final semantics, including translation validation of optimizing compilers, program verification, symbolic execution and processor hardware validation.

While this semantics still does not represent every aspect of an x86 processor, it remains the one of the most comprehensive and complete semantics of x86 currently in existence. In this paper we will address its most serious flaw - the lack of a formalization for x86 instruction decoding.

2.2 X86 DECODER - MOTIVATION

While the existing x86_64 semantics were a great step in the right direction, they still had one key flaw. They executed assembler mnemonics from a non-compiled assembly file. This format has several key drawbacks. First, it is not a particularly common or natural artifact to build during a normal project. The overwhelming majority of programmers will write code in higher level languages and then compile them down to machine code. And, most programs compiled to lower level languages will compile directly to binary code, rather than to this intermediate step. Furthermore, given that well studied semantics already exist for higher level languages, it would seem odd to compile such programs partially and then perform formal verification over the result rather than performing such verification over the source code.

Perhaps the compiler is untrustworthy - indeed the original purpose of the x86 semantics was translation validation from LLVM to x86. However, if the compilation architecture is not part of the trusted code base, then it seems bizarrely arbitrary to trust the assembler. Certainly, x86 assembly might be less complex than the compilation of high level source languages, but it's still non-trivial, especially if optimizations are performed during the assembly process. Thus, the inability for the semantics to cope with binary code would seem a serious drawback to its practical utility. Additionally, one must consider the case of verification over programs whose source code is unavailable, whether due to age or intellectual property concerns.

In spite of these drawbacks, it would seem seriously inefficient to reinvent the wheel by creating yet another semantics of x86 over just the binary format rather than leveraging the existing semantics. Thus, we find we need to translate from the binary format of x86 to an assembler mnemonics format that the existing semantics understands. The solution, then,

is disassembly.

This leaves two possible options – the first would be to use an external program to disassemble a binary before performing verification, and the second would be to integrate disassembly into the semantics of the language. The first option is unsatisfactory because x86 disassembly is highly nontrivial. If we wish to avoid adding an assembler to the trusted code base, it would seem useless to just replace it with a disassembler. While there exist a number of algorithms which disassemble x86, with even the simplest working for the majority of programs, none of them can offer any formal guarantee of correctness.

Indeed, since x86 programs can (in principle) self modify, or jump to arbitrary offsets which may overlap other possible sequences of instructions, it follows that in general the problem of disassembling an x86 binary is undecidable. Even on less pathological binaries, the relatively simple decoding algorithms supported by external tools cannot offer any correctness guarantees. As a result, we decided to implement an x86 decoder as a formalized part of the x86 semantics.

We determined that the best way of formalizing decoding was to port an existing implementation to the K framework, rather than reinventing the wheel. We had several reasons for this decision. x86 decoding is a well-defined algorithm, not an interesting unsolved problem. The sheer size of the instruction set precludes any form of manual implementation due to the sheer number of necessary rewrite rules, so we needed to find some input data to generate decoding rules. Finally, we realized the need to cross validate our solution against an existing implementation. As a result, we decided to implement the decoder by porting the Intel XED Disassembler into K. This program provided us with a well-tested decoding algorithm, input data and test standard. We then integrated this port into the existing x86 semantics, yielding a semantics of x86 capable of executing statically linked binaries.

2.3 CHALLENGES OF X86 DECODING

Before we discuss the actual x86 disassembler implementation, we will discuss the reason x86 decoding is so complex. As a CISC, x86 must encode an extremely wide variety of instructions, and must do so in a way that does not cause commonly used instructions to become too large for reasons of performance. Due to its commitment to backwards compatibility, it must furthermore accommodate a wide variety of older instruction sets added over the course of decades and designed around arcane and obsolete machine architectures. The net result of this historical baggage is that even the task of decoding a single instruction requires accounting for a great number of edge cases.

For example, in addition to the normal instruction decoding logic, the x86 disassembler

discussed here must also support three different methods of encoding vector extension instructions for SSE, AVX, XOP and similar instruction sets with alternate encodings. We furthermore made the decision to omit some of the more esoteric encoding schemes when we felt they would take an amount of effort disproportionate to their benefit, such as when we declined to support the AMD3DNow! instruction set. We made this choice based on the combination of its highly nonstandard encoding (placing the opcode where the decoder would expect to find a memory address displacement) and the fact that the instruction set has been obsolete for decades, and is not supported on modern processors. In other cases, such as Intel MMX instructions, we elected to keep outdated vector extension formats so that the decoder can work with the maximum possible number of programs, although the existing execution semantics does not always fully cover obsolete instruction sets. We leave this extension for future work.

As a result of all the non-standard formats, and additionally the sheer size of the x86 instruction set, while the algorithm is not overly difficult to understand, it remains highly complex. Even restricting ourselves to the 64 bit variant of x86 does not fully resolve these complexities. While x86_64 greatly simplified this situation by finally dropping support for some features, it does not eliminate the problem, and indeed has even added new instruction encodings that a modern disassembler must also account for. In the next section, we will describe the implemented disassembly algorithm in detail. We will first discuss the common case, and then briefly touch upon some of the alternate paths that varying alternate instruction encodings can take. The decoder formalization implements this algorithm in the K programming language as rewrite rules.

2.4 X86 DECODING ALGORITHM

This section will describe encoding of an x86 instruction, and how it applies the x86 decoding algorithm implemented by XED [5] and the K instruction decoder. Supplementary information on encoding formats is provided by [8] and [9].

An x86_64 binary instruction consists of between one to fifteen consecutive bytes in memory. These bytes are separated out into several different regions, each of which may have different sizes, or be absent entirely, in different instructions. Unfortunately, this implies that the decoder cannot know the size of the instruction it is decoding, or where the different regions of the instruction encoding are, until after the instruction has been determined, which occurs after most of these regions have already been decoded. Thus, the only correct algorithm is to scan bytes in order, passing them to a finite state machine, and decoding characteristics about the instruction at each step. This process continues until the decoder has

extracted a enough information to determine the precise instruction variant being encoded. With this information the decoder can determine the size and position of the remaining regions of the instruction and extract all other information in the encoding. It can then combine the directly decoded information from the instruction's binary with certain implicit information known about the instruction variant being decoded to create a complete description of the instruction.

The first region in any instruction is the prefix region, consisting of zero to four "prefix" bytes. The simplest of these are the legacy prefixes, which a programmer can prepend to most instructions to change certain characteristics of the instruction (for example, locking memory access or offering branch prediction hints), though some of these prefixes have become obsolete in x86_64 and are thus ignored by the decoder. Furthermore, in this prefix section the decoder may recognize one of several key prefixes (e.g. 0xC4, 0xC5) which indicate a specially encoded vector extension instruction. The detection of one of these vector extension prefixes causes the decoder to branch into a different decoding scheme since these prefixes create an entirely different space of possible instructions. For example, the 0xC4 prefix indicates that the next two bytes will be 0xC4 payload bytes, which contain information about the vector instruction.

We will discuss some of these special cases later. Aside from the prefixes signifying a non-standard encoding, the behavior of the x86 disassembler is simply to record the presence of these legacy prefix bytes. Some prefix bytes may be used in selecting an instruction variant later on, but otherwise we ignore them. The x86 semantics does not use a sufficiently complex execution or memory model to give many of the legacy prefixes meaning, for example: the x86 semantics does not simulate a branch predictor, so a branch hint prefix has no effect.

We deemed the extension of the semantics to effectively implement these prefixes as beyond the scope of this work. Such an implementation would require a simulation of the underlying hardware structures such as a branch predictor, a coherent cache structure, and the like. From our perspective reasoning about the behavior of x86 programs on an idealized uniprocessor machine, such implementation details are unnecessary complications - their implementation ought not to change the behavior of programs. In principle, capturing some behaviors of multithreaded programs could plausibly require such support, but we leave this for future work. Thus, in the present semantics, the prefix bytes region is primarily of interest due to the vector extension special cases, and the occasional instruction which uses the presence or absence of certain legacy prefixes to select between different variants.

The opcode region follows the prefix bytes. This consists of one to three bytes, and is the only region guaranteed to exist in any correctly encoded instruction. Note however that some obsolete encoding formats place it in non-standard positions, which would cause the

decoder to encounter it while processing other steps - we choose not to support such obsolete formats due to the small number of useful programs that would require them in the modern day compared to the large amount of effort it would take to support such alien encodings. In more standard x86 instruction formats, the encoding of the opcode region is relatively simple. Most of the time, the opcode region consists of a single opcode byte, which constitutes the instruction's entire opcode. Some exceptions to this rule include instructions which use certain vector extension prefixes, whose payload bytes may contain opcode extensions which change the precise behavior of the vector operation.

Additionally, for a few instructions, the opcode contains operand information (e.g. the INC instruction encodes the register to increment in the final three bits of its opcode byte). Such cases are relatively uncommon - however, it's not uncommon for the first byte of the opcode region to be one of several escape bytes. These bytes indicate a larger opcode, with the actual opcode value being in the second or third byte of the region. The spaces of opcodes created by differing lengths of the opcode region and differing escape bytes extending the region are called maps. The actual opcode and map are primary characteristics used to determine the instruction variant being decoded.

From the opcode, and the instruction's map, the decoder can determine the presence of the MODRM and SIB bytes. These single byte regions immediately follow the opcode, and encode information about the source and destination operands for the instruction (in the event that the instruction itself does not force certain operands, such as in the case of ret or push). The MODRM byte consists of three fields - MOD, REG and RM. The MOD field determines what memory addressing mode should be used for the instruction's memory operands. The REG field typically encodes a register operand, but some instructions use it as an opcode extension. Finally, the RM field specifies a register operand used to hold a memory address. Depending on the value of the MOD and RM fields, the instruction might or might not have a SIB byte.

The SIB byte contains the scale immediate and the index and base registers used to compute a memory address for a memory operand. Their encodings are relatively trivial - the decoder must simply read the byte and mask out the appropriate bits to determine the instruction operands.

In the event that the prefix region contains one of several vector extension prefixes, the decoder enters an alternate branch depending on the prefix. For example, the prefix 0xC4 indicates a two byte VEX prefix, and the prefix 0xC5 indicates a three byte VEX prefix. These following bytes contain additional information detailing how the instruction should treat its vector operands (such as the vector width), and opcode extensions to refine the selection of operation variants on these larger instruction sets. Furthermore, they contain

extensions to the MODRM's register selection fields which allow the instruction to address the additional vector registers used by these instructions. These vector instructions represent the long tail of x86 instructions – there are many competing vector extension encodings for instructions introduced over the course of years. These special cases then converge with the common case again after extracting their opcodes.

After extracting opcodes and MODRM/SIB presence, the decoder will have found enough information to determine the instruction being decoded. The K tool greatly simplifies implementing this instruction selection step, called the dynamic decode step. While the original XED algorithm uses an automatically generated multi-level hash table in C, requring multiple steps of automatically generated function calls, the K tool can simply use a single rewrite rule for each possible instruction variant. These rules are generated automatically and can be far more expressive than the original C implementation, since the K tool is natively capable of expressing all the required properties of each instruction variant and all of the resulting properties when the instruction variant is selected in a single, compact location. In the next section we will discuss how these rules are generated and give examples of generated rules.

After performing instruction selection, the remainder of the algorithm becomes relatively simple. During instruction selection the decoder sets values for the presence and size of the memory operand displacement and immediate operand bytes (as well as whether the immediate operand should be interpreted as a signed or unsigned value). These simply consist of little endian immediate values encoded directly in the instruction, whose length is determined by the instruction variant under encoding. Note that even these relatively simple regions have caveats to their encodings - for example, the ENTER instruction actually contains a three byte immediate region encoding two different immediate operands. However, for the majority of the instruction variants, the decoder can simply read in the displacement and immediate(s) as necessary.

After reading the immediate operand region, the decoder will have extracted all information in the binary instruction. At this stage, the decoder must generate an output. Initially, for testing purposes, we had the decoder output a string which could be compared with the output of XED on the same instruction. After having validated the decoder against the original XED implementation, we changed this step to output an x86 instruction K term, which could then be directly executed by the x86 semantics.

2.5 AUTOMATIC RULE GENERATION

The previous discussion of the x86 decoding algorithm handwaves one key problem: the sheer size of the x86 instruction set. To implement the dynamic-decode step, where the de-

coding algorithm chooses the correct instruction variant one would need at least one rule for each of the thousands of x86 instruction variants which match on the appropriate properties to determine the instruction, and set the instruction's implicit (i.e. implied by the nature of the instruction, rather enumerated specifically in its assembler or binary format) operands. Clearly, implementing thousands of such rules manually would be utterly impractical and extremely error prone. To solve this problem, we implemented rule generation scripts which digest XED input data files describing each instruction variant's binary format and generates K rules which implement the dynamic decode step for that instruction. Although extremely large, the resulting automatically generated K module remains quite human readable, in stark contrast to XED's generated C files. A sample of such a generated rule, and the XED input which caused its generation, is provided below.

```
{
ICLASS    : FCOMP
ATTRIBUTES: NOTSX
CPL       : 3
CATEGORY  : X87_ALU
EXTENSION : X87
FLAGS     : MUST [ fc0-mod fc1-mod fc2-mod fc3-mod ]
PATTERN   : 0xD8 MOD[mm] MOD!=3 REG[0b011] RM[nnn] MODRM()
OPERANDS  : REG0=XED_REG_ST0:r:IMPL:f80 MEM0:r:mem32real
            REG1=XED_REG_X87POP:r:SUPP
            REG2=XED_REG_X87STATUS:w:SUPP
}


// UNAME:
rule <REG0> _ => REG_ST0  </REG0>
<MEM0> _ => 1  </MEM0>
<REG1> _ => REG_X87POP  </REG1>
<REG2> _ => REG_X87STATUS  </REG2>
<k> DynamicDecodeInstruction => MODRM ~> ScanForDisp ~>
    ScanForImmediate ... </k>
<ICLASS> _ => FCOMP  </ICLASS>
<INUM> _ => 3  </INUM>
<CATEGORY> _ => "CATEGORY_X87_ALU"  </CATEGORY>
<INAME> _ => "fcomp"  </INAME>
```

```
<ATTRIBUTES> _ => ListItem(A_NOTSX)  </ATTRIBUTES>
<OPERANDS> _ =>
  ListItem(OperandMetadata(OPERAND_REG0,false,IMPLICIT,
                           xed_reg_enum_t,80,80,80))
  ListItem(OperandMetadata(OPERAND_MEM0,false,EXPLICIT,
                           xed_bits_t,32,32,32))
  ListItem(OperandMetadata(OPERAND_REG1,false,SUPPRESSED,
                           xed_reg_enum_t,AUTO,AUTO,AUTO))
  ListItem(OperandMetadata(OPERAND_REG2,false,SUPPRESSED,
                           xed_reg_enum_t,AUTO,AUTO,AUTO))
</OPERANDS>
<dynamicDecoderBuffer> 216 _:Ints  </dynamicDecoderBuffer>
<REG> 3 </REG>
<MOD> I1:Int </MOD>
<VEXVALID> 0 </VEXVALID>
requires I1 =/=Int 3
```

The generated rule checks for the opcode byte D8 with a MODRM byte whose REG value is three and whose MOD value is not three and that the decoder is not reading a vector extension operation, and from these properties concludes that the encoded instruction is the x87 fcomp instruction. It therefore sets several properties such as the instruction name and operands (including implicit operands such as the x87 status register) and schedules the decoding of a displacement and an immediate (though in this case no immediate will be found). Once this decoding step has completed, the decoder can trivially find the remaining data about the instruction.

While the dynamic decode step is the most significant instance of automatic rewrite rule generation, it is not the only one. Earlier on in the decoding process, the decoder makes use of automatically generated function rules to determine properties such as the presence or absence of a MODRM byte. This property must be extracted before the dynamic decode step can be completed, since the dynamic decode step needs to use properties from the MODRM byte to choose the precise instruction variant under decoding (for example, distinguishing between a form an instruction that reads from registers versus one that reads from memory), and hence requires a different generated function table.

2.6 SEMANTICS INTEGRATION

Of course, a semantics of x86 instruction decoding in isolation is of extremely limited practical value. To make use of these semantics, we next needed to integrate them into the existing execution semantics of x86.

After decoding has completed, the disassembler uses another family of automatically generated function rules to translate the decoded properties into a K term representing an x86 assembler mnemonic in AT&T syntax, the format accepted by the preexisting x86 semantics. This step, implemented only after the decoder had been tested against XED, created several technical issues that we needed to address before we could integrate the decoder semantics into the execution semantics.

The first serious technical issue was a difference in instruction encodings between the decoder semantics and the execution semantics. As mentioned earlier, the decoder's implementation was based on Intel's XED tool. The x86 execution semantics, on the other hand, were based upon the GNU assembler. In theory, XED is capable of emitting AT&T syntax, which GAS accepts. In practice, however, we found differences between the two implementations of x86 AT&T syntax. For example, certain instructions had different assembler mnemonics. Others encoded their operand sizes differently (operand sizes in x86 AT&T syntax are encoded using suffixes appended to the opcode by a process XED did not fully implement at the time of the decoder's creation). This compatibility issue made translating decoded instructions into K terms for the execution semantics a nontrivial task.

Ultimately, this problem was solved by creating a lookup table to translate between XED instruction variants and GAS opcodes. We created this lookup table by generating an assembler file consisting of every operation supported by the x86 semantics, assembling it using GAS and disassembling it using the K decoder. By matching up the emitted instruction variant number with the location of each instruction, we created a mapping from instruction variants emitted by the decoder to opcode terms accepted by the execution semantics. This mapping needed to be extended with the operand sizes of the instruction variants, as otherwise the mapping from XED instruction variants to GAS opcodes would not be a function. However, after handling this complexity generating the remainder of the K term by producing a representation for the operands themselves was relatively trivial.

In addition to this compatibility issue, we additionally needed to solve several more technical problems before we could claim to have produced a usable execution semantics of x86 binaries. To begin with, we needed to work around several limitations of the K language. The K Framework, having been designed to create semantics for high level languages, did not at the time of writing support accepting programs in a binary format.

This was solved relatively easily using an external parsing tool we wrote to extract the loadable segments of the ELF binary file being executed, and translate them into a format that K could accept. Choosing this format in particular caused us additional difficulties. The original x86 execution semantics represented memory regions as lists of lists (representing offsets in non-overlapping regions of memory). This model was chosen originally for compatibility with the K semantics of LLVM for the original translation validation project.

Unfortunately, this model simply did not scale up to the sizes required for binary execution. Many of the binaries we wished to execute were statically linked against glibc or another implementation of the C standard library. The text segment of the library code, once loaded into memory takes up several megabytes of space. The list of lists format used by the original execution semantics cannot represent this much data - K cannot handle lists this long due to recursion depth limits. Furthermore, we considered the original memory model more complex than necessary for a memory model of x86. Aside from compatibility with the LLVM K semantics, we found few reasons to maintain the old model.

So, as part of the integration with the decoder semantics, we rewrote the x86 execution semantics' memory model into a new two "layer" approach. Upon loading a new binary to execute, in order to accommodate the massive size of the glibc text segment, we store the initial contents of the new programs' memory as large hexadecimal strings which we substring and translate to machine integers upon demand. However, since that process is slow and does not offer a natural way of implementing memory writes, we also have another layer of memory.

The upper layer consisted of a mutable K map which contains the memory that has actually been read from or written to during the execution of a program. This map will be much smaller than the text segment, since much of the text segment is not actually used during a given execution (since the entire glibc library is linked into the binary, and much of it might not be used). The map section will also be faster to access than the hex string, and permits a natural implementation of writing to memory.

To perform an access, we first check to see if the address is contained in the map layer of memory. If it is, then we simply perform a read or write to the map. Otherwise, we check each loaded segment to see if the address falls inside any of the hex strings. If it does, we perform a substring operation to extract the relevant bytes from the hex string, and transform the result into an integer to return to the execution semantics. We furthermore cache the read in the map layer so that future reads to the same location will complete more quickly. This new memory model allowed for much faster loading times on binaries, and allowed us to handle loading even large binaries into memory.

2.7 EVALUATION

Initially, the decoder was validated independently of the execution semantics by full program disassembly. We implemented test code outputting disassembler strings, which were then compared against the output of XED on the same program, modulo whitespace differences. [10] After integration, we tested the full x86 binary execution semantics end-to-end by executing test programs. The original x86 execution semantics used a subset of the gcc torture test suite. However, there were some complexities with this. The semantics presently cannot execute system calls. The normal glibc initialization code makes several system calls for a variety of purposes. Hence, we replace the normal glibc implementation with our own minimal version. This includes initialization and deinitialization code.

In particular, we replaced the normal exit and abort system calls with our own non-standard methods of signalling the semantics to halt. We considered a test program to have executed successfully if it runs to completion and then exits normally, rather than crashing, getting stuck in an unfinished K configuration, or invoking abort after failing a self-test. We considered this an acceptable method of testing the decoder, since the actual instruction execution semantics had already been tested in their own paper.

Note that some programs in the original subset failed even when executed on native hardware - and thus were removed from consideration. As some of the tests are quite lengthy, we chose to run each program under a timeout. Of the 498 tests that executed successfully natively, 468 (97.1%) executed successfully under the decoder semantics. The failures were a mixture of programs that timed out, executed unsupported instructions, or called abort during execution.

2.8 CONCLUSION & LIMITATIONS

Presently, the semantics still have a number of areas for future improvement. They currently do not support dynamic linking at all, except detecting that they have been passed a dynamically linked binary and failing gracefully. This could be solved by either implementing a basic dynamic linker to perform this step before passing its result to K, or alternatively by using one of the several tools which transform a dynamically linked binary into a statically linked one before passing it to the existing semantics (though this option requires one to trust the transformation tool, or to use the final statically linked binary).

The next opportunity for future work involves writing a semantics of the operating system. Presently the x86 execution semantics offers definitions only for user mode instructions, and has no representation of any operating system at all. This is because, in principle, the

behavior of the operating system is an entirely different topic than the behavior of x86 itself. Still, the complete inability to perform system calls seriously limits the types of programs that can be executed on the semantics. This inability furthermore creates certain technical requirements as the semantics must jump to a fixed symbol, such as 'main' rather than starting at the normal ELF file start point in order to bypass normal library initialization code.

In conclusion, although much work remains to be done to create a fully featured semantics of the x86 programming language, in formalizing a semantics of x86 instruction decoding we have greatly expanded the space of programs that can be effectively analyzed with the x86 execution semantics. In doing so we have addressed a significant weakness in the original work created an opportunity for future improvement. For example, the decoder can reduce the complexity of the execution semantics by allowing for many existing execution semantics rules which cover different variants of the same instruction (such as an add that reads from registers rather than memory) to be coalesced into one rule.

Care will need to be taken, as this style of rule generalization is not always correct, but in many cases such optimizations offer the opportunity to greatly simplify the execution semantics, whose sheer size and complexity have thus far frustrated their use. We have furthermore demonstrated the practicality of using the K framework to analyze x86 binaries and other such low level machine languages in addition to the usual high level languages. In the next section, we will discuss the implementation of another machine language, Michelson, this time a blockchain smart contract implementation language rather than a physical hardware language.

CHAPTER 3: A FORMAL SEMANTICS OF TEZOS-MICHELSON

The Tezos blockchain, as described in its whitepaper [11] is a proof-of-stake based cryptographic ledger. Its has the unique property that its stakeholders (i.e. the owners of the Tezos currency, Tez) can amend the protocol to change many characteristics of the payment system. The amendable section of the protocol includes what constitutes a valid transaction on the Tezos blockchain, and how miners decide which candidate should become the next block on the chain, and so on. In addition to normal accounts, the Tezos blockchain supports accounts tied to smart contracts - programs hosted on the blockchain which can receive Tez, perform computation and then send Tez, originate new contracts, and so on.

Michelson [12] is the low level implementation language of these smart contracts. It is statically typed, fully deterministic and stack based, and contains many high level data structures. Like other low level languages, it exists primarily as a compilation target for higher level languages (e.g. Liquidity), and intentionally lacks some of the features a programmer might expect from a modern language. It avoids polymorphism and requires strict type checking before allowing contract origination. The type of the program stack can be determined statically at every code location in any contract. Contract entry and exit occurs at fixed points, and contract execution may only depend on the contract's stored data, a parameter passed to the contract when it is invoked, and some description of the present state of the block chain when the contract was called. Michelson lacks even a concept of named functions within a contract. And, although contracts may invoke other contracts, this can only be done by emitting token (Tez) transfer operations upon normal termination.

Unlike other low level languages, Michelson offers a number of high level language features such as complex data structures (maps, sets, and lists), sum types, product types, option types, unlimited precision integers, and so on. Michelson furthermore offers type and data annotations for both documentation purposes and to further refine typing information by distinguishing between different applications of the same type. Michelson contracts also have a concept of gas, which limits the length of computation permitted to a contract before it must terminate. Furthermore, Tez costs are assessed and paid to the contract executor for storage costs.

Note that, like other parts of the Tezos protocol, the implementation and semantics of Michelson instructions are subject to change through the normal amendment procedure, and, indeed the language has gone through several versions since Tezos' inception - notably adding methods of having multiple entry points to the same contract and thereby allowing contracts to have conceptually more than one parameter type (though this change was

implemented in a backwards compatible fashion so that older tools remained functional without modification).

A particular contract execution has no method of receiving new information from the outside world once it begins. In particular, this implies that Michelson contract execution is a pure function of the contract's storage, parameter and the current block chain state.

These restrictions exist to make Michelson execution predictable and ease formalization. In addition to the formalization discussed in this paper, a number of other tools have been created to serve the Tezos ecosystem. In the next section, we will discuss a prior formalization of the Michelson language before introducing our own formalization in the K Framework, known as K-Michelson. This formalization was produced in cooperation with Runtime Verification Inc. employee Stephen Skeirik, with further contributions from Runtime Verification Inc. employee Dwight Guth.

3.1 PRIOR WORK: MI-CHO-COQ

The closest work to our own is a formalization of Michelson in the Coq proving system called Mi-Cho-Coq [13]. It represents a Michelson contract as an inductive type over the types of individual Michelson instructions, which ultimately construct a type for the entire Michelson contract of the format (reproduced from the original Mi-Cho-Coq paper):

```
instruction ((pair parameter storage) :: nil)
            ((pair (list operation) storage) :: nil)
```

In which parameter and storage are type variables determining the type of the overall contract, and list operation are type literals. A consequence of this representation is that only well typed Michelson contracts are even possible to represent in their Coq format. They provide an interpreter which evaluates a contract to an error monad, and a weakest precondition verification scheme.

In comparison to our implementation, Mi-Cho-Coq has a type system whereas the K Michelson implementation is typeless, though their type system does not support annotations at the time of writing. Since Coq does not permit non-terminating functions[14], they use a depth parameter to enforce termination by way of an evaluation depth limit. In contrast, K permits non-termination in its rewrite rules, so we require no such addition. Neither Mi-Cho-Coq nor K-Michelson formalize the Michelson gas semantics.

3.2 K-MICHELSON

The K implementation of the Michelson language consists of slightly over three hundred rules. It makes the traditional design choice in K to implement a statically typed language as an un-typed execution semantics such that a type system can be added in a modular fashion at a later date. Initially, the semantics defined an ad-hoc representation of a Michelson contract combined with its parameter value, storage value and block-chain state - the four components which determine the result of a contract execution.

Separately, we defined an internal unit testing format which executed subsections of Michelson contracts in order to better verify the semantics. Since the semantics passed its initial versions, however, it became necessary to converge on a shared format between the K-Michelson semantics and the official Tezos Michelson implementation. This uniformity permits both implementations to execute shared test cases, and greatly improves our confidence that the K-Michelson formalization implements the same semantic behaviors as the official implementation.

3.2.1 Syntax

The concrete syntax of Michelson is called Micheline. This syntax is relatively simple. A sample of this syntax is included in the appendix. The basic syntactic element is a primitive application, which consists of a keyword identifier, such as 'option,' 'ADD,' 'contract' and so on, followed by zero or more arguments. These arguments may be other primitive applications, sequences of primitive applications (delimited by semicolons and enclosed in {} except for the outermost group), or one of the several forms of constants: True/False for booleans, unbounded integers constants, string constants, and hexadecimal value constants.

Although technically not excluded by the syntax, there exist requirements on the types of arguments allowed in any particular primitive application. An instruction is a form of primitive application, taking arguments depending on the particular instruction (e.g. the IF instruction accepts two sequence arguments containing the true or false branch of the conditional). A contract is then a sequence of three primitive applications of the form 'code { *Instruction Sequence* }' 'parameter *Type*,' and 'storage *Type*.' However, a contract's source code does not suffice. As alluded to earlier, one needs a value for its storage and parameter, as well as information about the state of the Tezos block chain, in order to execute a contract.

Presently, the semantics accepts as its input format a slight variation on the uniform unit testing format defined jointly between the Tezos foundation and the Runtime Verification team (described in more detail in a later section). This format (unlike the prior ad-hoc input)

follows the normal Micheline syntax - which is to say it consists of an unordered sequence of primitive applications. For example, the 'contract' primitive application contains the actual contract source code, and 'parameter_value' and 'storage_value' applications contain the contract's input data.

3.2.2 Semantics

Loading

The loading subsystem works by reading the outermost sequence of primitive applications, called "Groups." Since the semantics accept either a full contract or a unit test as input, the loading system must tolerate either format. Furthermore, per the documentation on the shared unit testing format, the order of the groups must not matter, and most groups must be optional (with certain sensible default values used in their absence). The possibility of a group's absence is handled simply by prepopulating the configuration with the default values.

The lack of ordering is more troublesome because the correct interpretation of some groups requires information from other groups to have already been loaded. For example, the correct interpretation of the 'parameter_value' group requires the type of the parameter (specified in the 'parameter' group) to already be known. But, the fix is relatively straightforward: we define a standard ordering of groups and sort the input groups before processing any of them. Fortunately, while some groups depend upon other groups, this dependency graph has no cycles. Most groups are extremely simple, for example 'now [Integer]' specifies the time returned by the Michelson NOW instruction. A few, such as the 'other_contracts' and 'Big_maps' groups are somewhat more involved because they specify more complex structures.

The most complicated single task during this loading section is translating Michelson Data from the Micheline input format to K's inner representation. For example, a dictionary in Michelson is specified as a sequence of primitive applications of the format { Elt K1 V1 ; Elt K2 V2 ; ... }. Keeping the map in this format would not benefit from the performance improvements from K's hooked Map sort, and would require reimplementing basic dictionary operations over this data type.

Instead, we define a function #ConcreteArgToSemantics which accepts a Micheline data element and a Michelson type and transforms the input element into a K representation of the same data. The type is also required because the literals for many sorts (such as maps and lists, or ints and nats) may be indistinguishable. This function also attaches type information

to input data when necessary to indicate which variant of an instruction will need to execute on that data (for example, 'ADD' on two integers has fundamentally different semantics than 'add' between to mutez values). It also does some sanity checks before returning converted data, allowing for quick detection of malformed inputs (for example, checking to ensure an constant of type natural is nonnegative) - though these checks are far from complete.

Finally, the last group processed will be the 'code' or 'contract' group, depending on whether this input file contains a unit test or a full contract. When the semantics encounters this group, it simply extracts the contained code and hands over execution to the main formal semantics.

Execution

Upon loading the code or contract group, the semantics switches into execution mode. During this section, the only mutable parts of the configuration are k cell and stack cell. While the loading section of the Michelson semantics necessarily contained a large amount of translation code to convert between Michelson data structures and K data structures, this section attempts to match the Michelson formal documentation as closely as possible. For instance compare the code implementing the DUP instruction in K:

```
rule <k> DUP A => #HandleAnnotations(A) ... </k>
     <stack> X:Data => X ~> X ... </stack>
```

With the formal specification of this rule in the Michelson documentation[12]:

```
DUP / x : S  =>  x : x : S
```

With the exception of the HandleAnnotations invocation, the rules are nearly identical. Allowing for this close matching was a major design goal of the Michelson semantics. Matching the Michelson formal specification as closely as possible allowed us to feel confident that we were implementing the correct behavior, because we could clearly see that the two forms of the rule are identical.

Some rules, however, are not as simple to match. For instance, many rules in the Michelson specification document use evaluation in conditionals. For example, the rule for map iteration:

```
> MAP body / {} : S  =>  {} : S
> MAP body / { Elt k v ; <tl> } : S  =>  { Elt k v' ; <tl'> } : S''
    where body / Pair k v : S  =>  v' : S'
    and MAP body / { <tl> } : S'  =>  { <tl'> } : S''
```

The recursive case of this rule specifies the behavior MAP in terms of itself on a shorter list. K does not offer a natural way of expressing this property, so we transform this recursive definition into an equivalent iterative one:

```
rule <k> #PerformMap(M1, M2, B) => B ~> #PopNewVal(#MinimalKey(M1))
        ~> #PerformMap(M1[#MinimalKey(M1) <- undef], M2, B) ... </k>
     <stack> . => Pair #MinimalKey(M1) {M1[#MinimalKey(M1)]}:>Data
             ... </stack>
     requires size(M1) >Int 0
```

The initial rewrite to transform the MAP instruction into the PerformMap operation is omitted for brevity, as is the base case. Note that there are subtleties to this implementation - Maps in Michelson are ordered data structures. Thus, when we perform a MAP operation, we must process the elements in ascending order. We accomplish this by repeatedly removing the minimum element (this implementation was chosen for simplicity, not speed). In cases like these, where we cannot simply directly translate the Michelson specification rule into K, the use of extensive unit testing becomes paramount to ensure that our formalization matches the behavior of the official implementation.

This testing became a project in its own right, and helped to uncover a number of bugs both in the semantics and in the official implementation. For instance, the formal specification rule above indicates that the block executed by the MAP instruction may modify stack values below the map element. We found during our cross testing of this instruction that this behavior did not work; instead any modification to the stack below the map made by the mapped block would be silently reverted when the MAP instruction completed execution. The Tezos foundation confirmed this bug and has issued a fix in a later version of Michelson.

3.2.3 Unit Testing & Cross Validation

The Michelson semantics implements the .tzt unit testing framework as specified by the Tezos foundation in cooperation with Runtime Verification [15]. A unit test in this format consists of a semicolon delimited sequence of input groups. All blockchain state related input groups such as 'big_maps' and 'other_contracts' are identical to the full contract format. The new groups consist of:

1. code: The instruction or block to execute.

2. input: The input stack to the instruction sequence. Note that, since this group represents a mid-execution state, this group permits internal syntax items such as operations which are forbidden as inputs to real contracts.

3. output: The final stack expected when the instructions in 'code' complete execution. This may be another mid-execution stack, like the input group, or it may be a failure case in the event that this test is expected to terminate abnormally. Additionally, this stack may contain '#Any' wildcards to specify that any value is acceptable in a certain position. This wildcard is useful for instructions such as 'CREATE_CONTRACT' which produce an unknown address as output, since the formal specification of that instruction does not indicate how Michelson implementation will chose the new address, we conservatively estimate it may chose any address for the newly originated contract.

The unit testing format is intentionally declarative. This permits the same test set to be reused for many different implementations or formalizations of Michelson. However, this also implies that the implementation under test must verify for itself whether its own output matches the expected output. This implies that a broken implementation might incorrectly pass itself on a unit test it should have failed, if it also incorrectly implements the equivalence check. We consider this a relatively minor problem, however, since checking equality in Michelson is a much simpler task than executing Michelson contracts, and an implementation with a broken equality check can furthermore be detected by running intentionally incorrect unit tests and observing that they fail.

An example unit test is included below:

```
code { MEM } ;
input { Stack_elt string "baz" ;
  Stack_elt (big_map string nat) 0 ; Stack_elt (big_map string nat) 0 } ;
output { Stack_elt bool False ; Stack_elt (big_map string nat) 0 } ;
big_maps { Big_map 0 string nat { Elt "bar" 4 ; Elt "foo" 11 } }
```

We cross-validated the Michelson semantics by first writing 116 unit tests in this format (actually translated from the original ad-hoc testing format), running them on both the K-Michelson semantics and the official Tezos Michelson implementation. It was during the initial creation of this test suite that we detected the 'MAP' instruction bug. We later augmented these original 116 unit tests by translating an additional 280 test cases from the official Michelson implementation's unit test suite into the new .tzt format. Since, at the time of writing, the official implementation does not yet implement support for actually

running these unit tests, we write a compatibility script layer to translate the .tzt format into a form that the official implementation can accept. This compatibility layer has certain limitations due to the limited command line interface offered by the Tezos implementation for testing. However, for the overwhelming majority of the unit tests, this format allows us to perform automatic cross validation between the K-Michelson semantics and the official Tezos implementation, and revealed both a number of discrepancies (which were subsequently fixed) and gave us the confidence needed to claim that K-Michelson faithfully implements the Michelson language.

The compatibility script needs to solve a number of issues in order to execute a .tzt script. The first issue is that the official tezos implementation only accepts full contracts for test execution, and offers no way of starting a contract part of the way through, and consequentially has no way of specifying an initial stack. As a consequence, the compatibility script must expand the test code into a full contract (filling in types for its parameter and storage), and must generate code to fill the input stack. This is accomplished by creating a template contract and generating a sequence of 'PUSH' instructions which set up the input stack. This expansion process is implemented in another K definition extending the normal Michelson semantics. The key rule is included below:

```
rule #FillTemplateContract(PushBlock, CodeBlock, ParamType) =>
  parameter ParamType ; storage unit .AnnotationList ;
  code { DROP .AnnotationList ; PushBlock ; CodeBlock ; UNIT @exitToken ;
         FAILWITH .AnnotationList }  ;
```

Using a K definition to generate the expanded contract offer the key advantage that we avoid needing to specify the Michelson syntax a second time. It however requires that we specify K rules for unparsing Michelson (that is - for generating a string which has the appropriate parse from an abstract syntax tree). This process is relatively straightforward for Michelson, besides some subtleties where parentheses are necessary or forbidden.

The next challenge the compatibility layer needed to solve was extracting output from the expanded Michelson contract. This task is nontrivial because the command line interface does not offer a way to stop a Michelson script prematurely to inspect an output stack. The contract cannot emit its entire stack, since a Michelson list can contain only one type of value, and Michelson at the time of writing contains no dynamic types. The exception to these rules is during contract failure. If the contract executes the 'FAILWITH' instruction, then the test interface dumps the entire stack at that moment of execution. Thus, the solution we decided upon was to fail all unit tests with a special token ('@exitToken' in the included code). The compatibility script will detect an output with this token and

interpret that as the contract 'failing successfully' and producing a valid output. It can then read the contents of the output stack from the emitted crash dump. We additionally use the command line interface's typechecking command to find the type of the stack upon the termination of the contract, and combine these two lists of information in order to determine the complete state of the contract upon failure. The typechecking information is necessary because the type of Michelson data cannot be inferred from its value in all cases. For example, "tz1NwQ6hkenkn6aYYio8VnJvjtb4K1pfeU1Z" is a valid value for any of the following types: String, Address and Contract Unit. This output stack is simply the stack after the test code's completion plus one additional 'Unit' element which the compatibility script can simply discard. There are additional subtleties here related to the interpretation of non-exitToken related failures, which the compatibility script also supports, but the full description of these cases is omitted for brevity.

The next challenge the compatibility needed to address was all of the blockchain state related loading groups required by the .tzt format. These are isolated from the remainder of the unit test by another K semantics extending the normal Michelson semantics. This avoids any parsing ambiguities in the main compatibility script by using the K parser. However, simply extracting these groups is not the main technical challenge. According to the format, a unit test should be able to specify the output for the following instructions: 'NOW,' 'AMOUNT,' 'BALANCE,' 'SENDER,' 'SOURCE,' 'CONTRACT,' 'SELF[1],' and 'CHAIN_ID'. Furthermore it should be possible to specify the contents of any big_maps associated with the contract.

Several of these instructions are impossible for the compatibility layer to support. In particular, the test interface offers no way of specifying the output of the 'BALANCE' instruction (it is simply hard-coded to always return one fixed value for a test script). The same applies to the 'CHAIN_ID' instruction - the value is not parameterized in the test interface and manipulating its real value is impossible without changing the Tezos implementation, which would defeat the cross-validation purpose of the compatibility layer. Furthermore, the 'NOW' instruction, which indicates the timestamp of the last block of the chain, is technically possible but practically infeasible to implement - the only way to manipulate it would be to change the system time.

Of the remaining loading groups, 'AMOUNT', 'SENDER', 'SOURCE' and 'SELF' are relatively trivial to support, with one caveat which they share with the 'CONTRACT' instruction. The former four are parameterized on the command line interface we use for the compatibility script, and thus are essentially trivial to support. For 'CONTRACT,' While

[1]Both the type and address, specified separately

we can originate contracts to fulfill the requirements of these instructions (though doing so requires creating a sandboxed blockchain), there is an additional problem. All Michelson contract addresses are generated by an unspecified but functionally random process - presumably to avoid address collisions on the real blockchain. We cannot specify what address we want a newly originated contract to have, even on the sandboxed blockchain. As a result, the compatibility script cannot promise that the addresses in the existing unit test will exactly match the addresses of the generated contracts. The compatibility script therefore instead interprets contract addresses in the .tzt format as essentially symbolic values, and replaces them after contract expansion with new, concrete addresses that it receives when it originates the other contracts the contract under test will interact with. The guarantees made are that identical symbolic addresses will be replaced with identical concrete addresses, and different symbolic addresses will be replaced with different concrete addresses. This is the strongest practical promise that the compatibility layer can make, and suffices for virtually all cases.

The compatibility layer still has other limitations, however. It cannot implement the big_maps group (and indeed does not support big_maps of any sort) - since executing a contract through the test interface does not actually perform an origination. Without an origination on the blockchain, we cannot store any big_map in the contract except by producing it on the fly during the initial setup phase. That approach would work, but would likely miss bugs associated with loading and unloading big_maps, and thus miss most of the implementation of that data structure, since the fact that big_maps do not need to be fully loaded into memory is their entire purpose in the Michelson language.

After executing the generated contract, its output is combined with the expected output of the unit test and passed to yet another K definition. This K definition must then compare the actual and expected outputs and ensure that they match, accounting for any #Any wildcards in the expected value. The primary technical challenge here however are Blockchain Operations: Contract origination, token transfers and delegation changes. These internal operations were not designed to be exposed directly to the user. Hence, there is no concrete syntax for them. This has two consequences. The more minor consequence is that they cannot be added to an input stack - this does not pose a problem for the compatibility layer as no unit test needs to accept an input operation, since operations mostly cannot be manipulated by Michelson code. The bigger problem, however, is that the Michelson test interface outputs them in bytes format due to their lack of a concrete syntax. K cannot interpret this format directly. Thus, we needed to write a hook for comparing this binary form which uses the Michelson command line interface to transform the binary blob the test interface outputs into Json, which K can then parse and transform into a proper K term for

comparison.

In spite of these limitations, this compatibility script suffices to successfully execute more than 80% of the nearly 400 unit tests written during the course of this project. As all of these tests also pass on the K-Michelson implementation, they give us confidence that our formalization matches the official executable implementation, and, consequentially, that the official implementation faithfully implements the semi-formal specification we based our semantics on. An additional 22 'coverage' tests were additionally written. These tests mostly verify that the semantics behaves as expected (i.e. fails appropriately) on malformed inputs, or that it tolerates non-standard but well formed inputs (such as listing the sections of a Michelson contract in a different order). All of these tests combined yield full code coverage of the semantics (i.e. every rule is executed by at least one unit test). This gives us high confidence that the semantics behave as expected.

3.3 LIMITATIONS & KNOWN ISSUES & FUTURE WORK

Presently the main limitation of the semantics come from the lack of a specification for the 'PACK' and 'UNPACK' instructions in the Michelson documentation. Without a formal definition for these instructions, the semantics have no implementation for Michelson's serialization or deserialization procedures. Hence, we are unable to execute any contracts which use these instructions in a non-trivial way. For the moment, the only implementation we have is to axiomatize the UNPACK of data produced by PACK as an identity operation. This allows us to at least test these instructions on a basic level. However, most real contracts that make use of these instructions will obviously not simply use this identity operation, and thus we will be unable to execute them properly.

The next serious limitation at this moment is derived from the first. Without a meaningful way of using bytes due to the lack of serialization or deserialization operations, there has been little reason to implement many operations over bytes. In particular, the semantics do not presently implement the cryptographic operations Michelson offers (Blake2B, SHA256 and SHA512, as well as verifying a signature). Ultimately, however, we anticipate these being implemented with a hook to a cryptographic library. Presently, they are axiomatized.

As mentioned earlier, the semantics are presently only an execution semantics for the Michelson language. Hence, they do not perform typechecking and might execute an ill-typed contract. This was not deemed a serious issue as the official Michelson implementation already offers a convenient typechecking interface, and the existing semantics should be able to execute any well typed Michelson contract correctly without performing a separate typecheck. Or, so we thought. This leads into the main known issue with the semantics.

The only known true serious behavioral defect in the semantics comes from this very early design decision - the choice to separate out the execution semantics from the eventual typechecking semantics. This choice was made in the belief that for the most part changing the type of an instruction will not change the ultimate value produced by the instruction (except, perhaps, to change its type). For example, the 'ADD' over the ints 2 and 2 will result in an int of 4, and the same instruction over the nats 2 and 2 will result in a nat of 4. Thus, the type of the inputs doesn't change the values of the outputs.

As it turns out, one exception exists. The 'CONCAT' instruction, executed on a list of values, will combine all elements of that list. The CONCAT operation can work over lists of either strings or bytes. Most of the time, the semantics can distinguish the two cases by examining the elements of the list and choosing the version of 'CONCAT' that matches the types of the elements. However, in the event of an empty list, CONCAT must determine the type of the list without any elements. We currently record the type of a list when it is created, and thus can cover most of these cases as well. However, the 'MAP' instruction applied to lists may yield a list of another type. If this list is non-empty, then this does not pose a serious problem as we can determine the type of the list by its contents. However, consider the following code snippet:

```
{ NIL ??? ; MAP { Some Arbitrary instruction block } ; CONCAT ; ... }
```

No elements exist in the list of unknown type created by the 'NIL' instruction, so the 'MAP' block will never actually execute, and the MAP instruction will no-op. However, the correct variant of 'CONCAT' to execute at the end of the snippet will depend on the static type of the MAP's instruction block. Hence, executing this snippet correctly without performing a typecheck on the contract and determining the map's output type is impossible.

Hence, we believe that a reasonable next step to extend the semantics would be to add a static type system which annotates each position in the contract with its type. This would enable the semantics to correctly execute the previous confounding snippet, and may allow for more properties to be proven automatically using the semantics, as the prover could exclude behaviors that are ill-typed. We leave this for future work.

3.4 PERFORMANCE

Performance was not considered a major priority when writing the semantics. However, due to extensive work by the K team optimizing the LLVM backend, interpreters generated by K semantics for the LLVM backend sometimes outperform even purpose-built interpreters. In this section, we will briefly discuss the performance differences between the

K-Michelson interpreter and the official OCaml implementation of Michelson on a simple sum-to-N Michelson contract. The contract under test follows:

```
parameter_value N ;
storage_value 0 ;
contract {
  storage nat ;
  parameter nat ;
  code { LEFT nat ;
         LOOP_LEFT { DUP ;
                     DIP { CDR } ;
                     CAR ;
                     DUP ;
                     DIP { ADD } ;
                     PUSH nat 1 ;
                     SWAP ;
                     SUB ;
                     ISNAT ;
                     IF_NONE { RIGHT (pair nat nat) }
                             { PAIR ; LEFT nat } } ;
                     NIL operation;
                     PAIR } }
```

This contract computes the value of $1 + 2 + ... + N$. We compare the performance of the K-Michelson semantics by compiling it on the LLVM backend with full optimizations set, and using KServer - a tool included in K which avoids the need to repeatedly reload the JVM for the K Frontend, and caches certain parsing information.

We compare this to the official Tezos Michelson implementation running the normal "tezos-client run script ..." command on a sandbox node. Timing information is collected with the default 'time' command and real time is reported. Additionally, krun is run once before data collection begins so that KServer has already loaded the semantics (otherwise this loading would skew the results for the first data collection). The results are below:

Table 3.1: Performance Results Table

N	K (s)	Official (s)	K (New Parser, s)
1	1.512	0.056	0.084
300000	2.109	1.984	1.371
600000	2.625	3.928	2.592

More recently with the assistance of Runtime Verification, we have experimented with alternate parsers. In recent versions, K offers the ability to generate a bison parser to use as an alternative to its built-in KAST parser. We thank the team at Runtime Verification for their code contributions and advice in integrating these alternatives.

These improvements have offered a substantial speedup, particularly on short executions, but do not change the overall conclusion. For short contracts with little computation, the official implementation has a clear lead due to the overhead involved in starting up K. For longer contracts, however, the LLVM backend actually runs the contract more quickly, and so for sufficiently long contract executions K will overtake the official implementation.

To be fair, this comparison is not perfect. K-Michelson does not yet support some features that the official Tezos implementation does, such as static typechecking or gas recording. Furthermore, while K-Michelson supports expanding Michelson macros, we do not account for this step in the prior analysis since it involves invoking the official implementation, and could in principle be done offline before the contract actually needs to be executed. That said, we believe this case study makes the point that the K implementation of Michelson is sufficiently fast to be practical for contract testing and deployment.

3.5 VERIFICATION

While it is impossible to overstate the utility of being able to execute concrete Michelson code at comparable speeds to the official implementation, the primary purpose of the semantics is formal verification. In this section, we discuss the verification of the 'vote' contract from the Mi-Cho-Coq [13] paper. The entire behavior of this contract requires three claims, since the contract has one success and two failure modes. Due to the length of the contract and the consequent size of the claims, we will focus on the success case here, and include the rest of the claims in the appendix.

```
rule <k>
   storage_value ((M:Map)[S:String <- I:Int]) ;
   parameter_value S ;
```

```
 amount A:Int ;
 contract {
   storage (map .AnnotationList (string .AnnotationList)
                                (int .AnnotationList)) ;
   parameter (string .AnnotationList) ;
   code {
     AMOUNT .AnnotationList ;
     PUSH .AnnotationList mutez .AnnotationList 5000000 ;
     COMPARE .AnnotationList ;
     GT .AnnotationList ;
     IF .AnnotationList { UNIT .AnnotationList ;
                          FAILWITH .AnnotationList }
                          { } ;
     DUP .AnnotationList ;
     DIP .AnnotationList { CDR .AnnotationList ;
                           DUP .AnnotationList } ;
     CAR .AnnotationList ; DUP .AnnotationList ;
     DIP .AnnotationList {
         GET .AnnotationList ;
         IF_NONE .AnnotationList {
           UNIT .AnnotationList ;
           FAILWITH .AnnotationList } {} ;
           // ASSERT_SOME .AnnotationList ;
         PUSH .AnnotationList int .AnnotationList 1 ;
         ADD .AnnotationList ;
         SOME .AnnotationList
     } ;
     UPDATE .AnnotationList ;
     NIL .AnnotationList operation .AnnotationList ;
     PAIR .AnnotationList } } => . </k>
<stack> . => (Pair #List(.List, operation .AnnotationList)
             (M[S <- (1 +Int I)])) </stack>
<paramtype> #NotSet => ?_ </paramtype>
<paramvalue> #NoData => ?_ </paramvalue>
<storagetype> #NotSet => ?_  </storagetype>
<storagevalue> #NoData => ?_ </storagevalue>
```

```
<mybalance> #Mutez(0) => ?_ </mybalance>
<myamount> #Mutez(0) => #Mutez(A) </myamount>
<mynow> #Timestamp(0) => ?_ </mynow>
<myaddr> #Address("InvalidSelfAddr") => ?_ </myaddr>
<knownaddrs> .Map => ?_ </knownaddrs>
<sourceaddr> #Address("InvalidSourceAddr") => ?_ </sourceaddr>
<senderaddr> #Address("InvalidSenderAddr") => ?_ </senderaddr>
<mychainid> #ChainId(0x) => ?_ </mychainid>
<nonce> #Nonce(0) => ?_ </nonce>
<bigmaps> .Map => ?_ </bigmaps>
<returncode> 1 => ?_ </returncode>
requires A >Int 5000000 andBool A <Int 9223372036854775808
```

The claim takes the form of a partially symbolic full contract execution of the vote contract. Note that we have expanded the macros of the vote contract manually so that the code consists only of pure Michelson instructions.

The vote contract implements essentially an election. It stores a map of candidates, whose vote counts can be incremented by sending money to the vote contract. This claim specifies that, if an amount of mutez greater than 5,000,000 (5 Tez) is sent to the contract, along with a parameter string which is inside the map (and thus represents a valid candidate), then the result of executing the vote contract will be to replace the stored map with a copy in which the chosen candidate has gained one vote, and the map is otherwise unchanged, and to emit no operations.

The preconditions of this claim are specified in the first three loading groups of the contract, and the final requires clause:

```
storage_value ((M:Map)[S:String <- I:Int]) ;
parameter_value S ;
amount A:Int ;
...
requires A >Int 5000000 andBool A <Int 9223372036854775808
```

We use a symbolic map update over an arbitrary map M which inserts a symbolic int into the map using the parameter string S as a key to specify the precondition from the Mi-Cho-Coq paper - that the key S exists in the stored M. The other precondition on the amount passed to the contract is specified as restrictions on the symbolic value of A. Note

that, due to details of the K implementation, we also specify that A must be a valid Mutez value (and thus be less than 2^{63}).

Actually proving this claim is relatively simple, since the contract specified contains no loops or loop-like constructs from Michelson. Like the Mi-Cho-Coq paper, we require a few basic axioms about the Map data structure. However, since in our implementation Map is a K-construct, we can mostly ignore the Michelson map operations and focus on the language-agnostic K structure.

```
rule S:String in_keys ((M:Map)[S:String <- _]) => true [simplification]
rule ((M:Map)[S:String <- I:Int][S]) => I [simplification]
rule ((M:Map)[S <- _][S <- J]) => M[S <- J] [simplification]
```

The axioms themselves are mostly trivial - a symbolic map with an update adding a value for a key 'S' does in fact contain that key S. The value of a lookup of the key S in such a symbolic map is the updated value, and two updates to the same map on same key result in the rightmost one overwriting the updates to their left. With these rules axiomatized, the remainder of the proof can be done automatically. We feel this example, combined with the extensive cross validation testing against the official Michelson implementation, show that the Michelson semantics offer a useful tool for the formal verification of Michelson contracts.

3.6 COMPARISON TO X86 & CONCLUSION

We will close by discussing the stark differences between the different styles of machine language and how Michelson's design differs from x86's so as to simplify formalization efforts such as ours. To begin with, we will briefly compare the final product of both the x86 formalization efforts and the Michelson formalization efforts. The product of this Michelson project was a nearly complete specification of the execution semantics of the language, produced from scratch over the course of slightly over six months.

This should contrast quite starkly with the semantics of x86, which took well over twice as much time to formalize only a large subset, and that with extensive assistance from automatic generation and prior work. Furthermore, the difference in the utility of their outputs was similarly stark. The x86 semantics still needs extensive assistance in executing realistic programs due to the lack of system calls and sheer size of the execution model. Even when it does work, the performance of the x86 semantics is, unsurprisingly, many orders of magnitude slower than an actual processor.

The Michelson semantics can execute nearly any Michelson program, and does so at performance levels comparable to the original implementation. From the perspective of formal

verification, Michelson has many advantages over x86. The first one is size: Michelson is far a smaller language than x86. Its entire semantics has an order of magnitude fewer rules than the x86 decoder. The next is specification format - while Michelson's official specification is similarly semi-formal and non-executable, it is expressed in precise rewrite rules rather than imperative pseudocode. The third advantage is Michelson's lack of extensive side effects - where an x86 instruction may modify arbitrary registers not explicitly part of its operands, and status flags, and memory, a Michelson instruction will only modify some number of variables on the top of the stack.

Finally, where x86 has been implemented by hundreds of different processors, made by different companies over the years, Michelson has a single canonical implementation to test against. Ultimately, these are the key features that make Michelson a language well designed for formalization.

REFERENCES

[1] G. Rosu, *K: A semantic framework for programming languages and formal analysis tools*, 01 2017, pp. 186–206.

[2] C. Ellison and G. Rosu, "An executable formal semantics of c with applications," in *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. ACM, January 2012, pp. 533–544.

[3] D. Park, Andrei, and G. Ros, "A complete formal semantics of javascript," 2015.

[4] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, B. Moore, Y. Zhang, D. Park, A. Ştefănescu, and G. Roşu, "Kevm: A complete semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium*. IEEE, 2018, pp. 204–217.

[5] "intelxed/xed : x86 encoder decoder." [Online]. Available: https://github.com/intelxed/xed

[6] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "A complete formal semantics of x86-64 user-level instruction set architecture," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3314221.3314601 pp. 1133–1148.

[7] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified synthesis: Automatically learning the x86-64 instruction set," *SIGPLAN Not.*, vol. 51, no. 6, p. 237–250, June 2016. [Online]. Available: https://doi.org/10.1145/2980983.2908121

[8] B. Petkov, "X86-64 instruction encoding." [Online]. Available: https://events.static.linuxfound.org/sites/events/files/slides/bpetkov-x86-hacks.pdf

[9] "X86-64 instruction encoding." [Online]. Available: https://wiki.osdev.org/X86-64_Instruction_Encoding

[10] D. Miranti and Rosu, "Towards formalizing the x86-64 instruction decoder in k." [Online]. Available: https://www.cl.cam.ac.uk/~jrh13/spisa19/paper_03.pdf

[11] L. M. Goodman, "Tezos : A self-amending crypto-ledger position paper," 2014.

[12] "Michelson: the language of smart contracts in tezos." [Online]. Available: https://tezos.gitlab.io/whitedoc/michelson.html

[13] B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson, "Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts," *arXiv e-prints*, p. arXiv:1909.08671, Sep. 2019.

[14] "Mi-cho-coq readme." [Online]. Available: https://gitlab.com/nomadic-labs/mi-cho-coq/tree/master

[15] R. Cauderlier, "Wip: Doc/michelson: document a michelson extension for writing unit tests." [Online]. Available: \url{https://gitlab.com/tezos/tezos/-/merge_requests/1487/diffs}

APPENDIX A: SEMANTICS REPOSITORIES

The three K semantics discussed in this paper can be found at the following repositories:

1. The x86 decoder:
   `https://gitlab.engr.illinois.edu/miranti2/x86-disassembler`
2. The x86 semantics:
   `https://github.com/kframework/X86-64-semantics/tree/new_memory_model`
3. The Michelson semantics:
   `https://github.com/runtimeverification/michelson-semantics`

APPENDIX B: FULL VOTE CONTRACT CLAIMS FILE

```
requires "../../.build/unit-test.k"
requires "lemmas.k"
module VOTE
  imports UNIT-TEST
  imports LEMMAS
  rule <k>
         storage_value ((M:Map)[S:String <- I:Int]) ;
         parameter_value S ;
         amount A:Int ;
         contract {
           storage (map .AnnotationList (string .AnnotationList)
                                        (int .AnnotationList)) ;
           parameter (string .AnnotationList) ;
           code {
             AMOUNT .AnnotationList ;
             PUSH .AnnotationList mutez .AnnotationList 5000000 ;
             COMPARE .AnnotationList ;
             GT .AnnotationList ;
             IF .AnnotationList { UNIT .AnnotationList ;
                                  FAILWITH .AnnotationList } { } ;
             DUP .AnnotationList ;
             DIP .AnnotationList { CDR .AnnotationList ;
                                   DUP .AnnotationList } ;
             CAR .AnnotationList ; DUP .AnnotationList ;
             DIP .AnnotationList {
                 GET .AnnotationList ;
                 IF_NONE .AnnotationList {
                   UNIT .AnnotationList ;
                   FAILWITH .AnnotationList } {} ;
                 PUSH .AnnotationList int .AnnotationList 1 ;
                 ADD .AnnotationList ;
                 SOME .AnnotationList
             } ;
             UPDATE .AnnotationList ;
             NIL .AnnotationList operation .AnnotationList ;
```

```
        PAIR .AnnotationList } } => . </k>
      <stack> . => (Pair #List(.List, operation .AnnotationList)
                   (M[S <- (1 +Int I)])) </stack>
      <paramtype> #NotSet => ?_ </paramtype>
      <paramvalue> #NoData => ?_ </paramvalue>
      <storagetype> #NotSet => ?_  </storagetype>
      <storagevalue> #NoData => ?_ </storagevalue>
      <mybalance> #Mutez(0) => ?_ </mybalance>
      <myamount> #Mutez(0) => #Mutez(A) </myamount>
      <mynow> #Timestamp(0) => ?_ </mynow>
      <myaddr> #Address("InvalidSelfAddr") => ?_ </myaddr>
      <knownaddrs> .Map => ?_ </knownaddrs>
      <sourceaddr> #Address("InvalidSourceAddr") => ?_
      </sourceaddr>
      <senderaddr> #Address("InvalidSenderAddr") => ?_
      </senderaddr>
      <mychainid> #ChainId(0x) => ?_ </mychainid>
      <nonce> #Nonce(0) => ?_ </nonce>
      <bigmaps> .Map => ?_ </bigmaps>
      <returncode> 1 => ?_ </returncode>
      requires A >Int 5000000 andBool A <Int 9223372036854775808

rule <k>
       storage_value M:Map ;
       parameter_value S:String ;
       amount A:Int ;
       contract {
         storage (map .AnnotationList (string .AnnotationList)
                 (int .AnnotationList)) ;
         parameter (string .AnnotationList) ;
         code {
           AMOUNT .AnnotationList ;
           PUSH .AnnotationList mutez .AnnotationList 5000000 ;
           COMPARE .AnnotationList ;
           GT .AnnotationList ;
           IF .AnnotationList { UNIT .AnnotationList ;
                                FAILWITH .AnnotationList }
            { } ;
```

```
            DUP .AnnotationList ;
            DIP .AnnotationList { CDR .AnnotationList ;
                                  DUP .AnnotationList } ;
            CAR .AnnotationList ; DUP .AnnotationList ;
            DIP .AnnotationList {
                GET .AnnotationList ;
                IF_NONE .AnnotationList {
                  UNIT .AnnotationList ;
                  FAILWITH .AnnotationList } {} ;
                PUSH .AnnotationList int .AnnotationList 1 ;
                ADD .AnnotationList ;
                SOME .AnnotationList
            } ;
            UPDATE .AnnotationList ;
            NIL .AnnotationList operation .AnnotationList ;
            PAIR .AnnotationList } } =>
            #FindVerifyOutput(., Aborted(?_, ?_, ?_, ?_)) </k>
        <stack> . => ?_ </stack>
        <paramtype> #NotSet => ?_ </paramtype>
        <paramvalue> #NoData => ?_ </paramvalue>
        <storagetype> #NotSet => ?_  </storagetype>
        <storagevalue> #NoData => ?_ </storagevalue>
        <mybalance> #Mutez(0) => ?_ </mybalance>
        <myamount> #Mutez(0) => #Mutez(A) </myamount>
        <mynow> #Timestamp(0) => ?_ </mynow>
        <myaddr> #Address("InvalidSelfAddr") => ?_ </myaddr>
        <knownaddrs> .Map => ?_ </knownaddrs>
        <sourceaddr> #Address("InvalidSourceAddr") => ?_
        </sourceaddr>
        <senderaddr> #Address("InvalidSenderAddr") => ?_
        </senderaddr>
        <mychainid> #ChainId(0x) => ?_ </mychainid>
        <nonce> #Nonce(0) => ?_ </nonce>
        <bigmaps> .Map => ?_ </bigmaps>
        <returncode> 1 => ?_ </returncode>
        requires 0 <=Int A andBool 5000000 >Int A

rule <k>
```

```
 storage_value M:Map ;
 parameter_value S:String ;
 amount A:Int ;
 contract {
   storage (map .AnnotationList (string .AnnotationList)
                                (int .AnnotationList)) ;
   parameter (string .AnnotationList) ;
   code {
     AMOUNT .AnnotationList ;
     PUSH .AnnotationList mutez .AnnotationList 5000000 ;
     COMPARE .AnnotationList ;
     GT .AnnotationList ;
     IF .AnnotationList { UNIT .AnnotationList ;
                          FAILWITH .AnnotationList } { } ;
     DUP .AnnotationList ;
     DIP .AnnotationList { CDR .AnnotationList ;
                           DUP .AnnotationList } ;
     CAR .AnnotationList ; DUP .AnnotationList ;
     DIP .AnnotationList {
         GET .AnnotationList ;
         IF_NONE .AnnotationList {
           UNIT .AnnotationList ;
           FAILWITH .AnnotationList } {} ;
         PUSH .AnnotationList int .AnnotationList 1 ;
         ADD .AnnotationList ;
         SOME .AnnotationList
     } ;
     UPDATE .AnnotationList ;
     NIL .AnnotationList operation .AnnotationList ;
     PAIR .AnnotationList } } =>
     #FindVerifyOutput(., Aborted(?_, ?_, ?_, ?_)) </k>
<stack> . => ?_ </stack>
<paramtype> #NotSet => ?_ </paramtype>
<paramvalue> #NoData => ?_ </paramvalue>
<storagetype> #NotSet => ?_  </storagetype>
<storagevalue> #NoData => ?_ </storagevalue>
<mybalance> #Mutez(0) => ?_ </mybalance>
<myamount> #Mutez(0) => #Mutez(A) </myamount>
```

```
        <mynow> #Timestamp(0) => ?_ </mynow>
        <myaddr> #Address("InvalidSelfAddr") => ?_ </myaddr>
        <knownaddrs> .Map => ?_ </knownaddrs>
        <sourceaddr> #Address("InvalidSourceAddr") => ?_
        </sourceaddr>
        <senderaddr> #Address("InvalidSenderAddr") => ?_
        </senderaddr>
        <mychainid> #ChainId(0x) => ?_ </mychainid>
        <nonce> #Nonce(0) => ?_ </nonce>
        <bigmaps> .Map => ?_ </bigmaps>
        <returncode> 1 => ?_ </returncode>
        requires A >Int 5000000 andBool
                 A <Int 9223372036854775808 andBool
                 notBool(S in_keys(M))
endmodule
```

APPENDIX C: MICHELSON SYNTAX EXCERPTS

```
syntax ContractGroup ::= "contract" "{" Contract "}"
syntax NowGroup ::= "now" Int
syntax SenderGroup ::= "sender" String
syntax SourceGroup ::= "source" String
syntax ChainGroup ::= "chain_id" MBytes
syntax SelfGroup ::= "self" String
syntax AmountGroup ::= "amount" Int
syntax BalanceGroup ::= "balance" Int
syntax ContractsGroup ::= "other_contracts" OtherContractsMap
syntax ParameterValueGroup ::= "parameter_value" Data
syntax StorageValueGroup ::= "storage_value" Data
syntax BigMapGroup ::= "big_maps" BigMapMap

syntax Group ::= ContractGroup
               | ParameterValueGroup
               | StorageValueGroup
               | NowGroup
               | SenderGroup
               | SourceGroup
               | ChainGroup
               | SelfGroup
               | AmountGroup
               | BalanceGroup
               | ContractsGroup
               | BigMapGroup

syntax Groups ::= Group | Group ";" Groups | Group ";"
syntax Pgm ::= Groups
```