Formal Verification Framework Based on Symbolic Execution for Smart Contract

Thi Thu Ha Doan
University of Freiburg, Germany

Peter Thiemann
University of Freiburg, Germany

Abstract

In the context of blockchain technology, the immutability of smart contracts once implemented underscores the critical need to ensure their accuracy. Even in cases where smart contract implementations are not overly extensive and have undergone testing before deployment, the blockchain community has identified significant vulnerabilities in their design. In addition, the relatively new nature of smart contract languages has led to unforeseen errors due to a lack of familiarity with their intricacies. To overcome these challenges, formal verification emerges as a key solution to guarantee the correctness of smart contracts. In response to this need, we have developed a formal verification tool for smart contracts, particularly those written in Michelson. This tool uses symbolic execution to simulate the implementation of the smart contract language, helping to detect subtle errors that are difficult for smart contract developers to detect. In addition, our tool includes a domain-specific language that allows users to precisely specify contract properties. By interacting with an SMT solver, it can handle a wide range of properties. In particular, it streamlines the process of reviewing requirements, uncovering hidden errors, and validating user-defined properties. In summary, our research highlights the need for robust verification of smart contracts. We present a purpose-built tool that utilizes symbolic execution and a domain-specific language to improve the correctness of smart contracts and provide a comprehensive solution to mitigate potential pitfalls in blockchain-based applications.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Smart Contract, Blockchain, Formal Verification, Symbolic Execution

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding *Thi Thu Ha Doan*: (Optional) author-specific funding acknowledgements
Peter Thiemann: [funding]

Acknowledgements I want to thank ...

1 Introduction

Symbolic execution techniques encounter challenges related to state explosion, particularly when applied to large systems. Fortunately, the constraints of storing and running smart contracts on a blockchain platform impose size limitations, making them manageable for symbolic execution.

In this work, we present the design of a domain-specific language for specifying smart contract properties and demonstrate the integration of symbolic execution with an SMT solver for formal verification of these properties.

2 Symbolic Execution Model

2.1 System Model

Michelson serves as the low-level smart contract language for the Tezos blockchain platform. The processing and manipulation of data during the execution of smart contracts are managed using a stack. The representation of a stack can be described as follows:

$[\,]$ represents the empty stack.

$hd :: tl$ denotes a stack with the top element as hd and the remaining stack as tl.

Each piece of data in Michelson is assigned a specific type, and the language restricts programmers from introducing custom types. The complete list of types can be found in Figure 3. To model the system, we use a stack in which each element is a pair consisting of term and its corresponding type. Terms are defined in Figure 4. The stack is denoted as $S = (t_1, ty_1) :: (t_2, ty_2) :: \ldots :: [\,]$ where elements are types paired with terms.

A Michelson program is a sequence of instructions executed in a sequential manner. Each instruction takes, as input, the stack produced by the preceding instruction and modifies it for the subsequent one. Let $I = i_1;\ i_2;\ \ldots\ i_n$ represent a sequence of instructions. During symbolic execution, the system state evolves, assuming different values based on the constraints of symbolic values. Moreover, branching may occur depending on conditions, and both constraints and branching conditions are recorded as predicates. Let P is a path predicate that is extended at each conditional and expressed in conjunction form, capturing the branching conditions and constraints encountered during the symbolic execution process.

▶ **Definition 1.** *A symbolic execution state is represented by the tuple ST = [I, S, P]. Here, SE = {ST_1, ST_2, ..., ST_n} range over sets of states.*

The smart contract program takes a stack as input, consisting of a single pair where the first element is an input value referred to as parameter denoted as *par*and the second element represents the content of the storage space denoted as *stg*. The program produces a stack as output, which is a single pair. The first element of this pair is the list of internal operations that the program intends to emit (opl), and the second element is the new contents of the storage space. Let's define the initial stack as S_{init} = (Pair $par\ stg$, pair $ty_1\ ty_2$) :: $[\,]$, where par and stg denote terms representing the parameter and the storage, and ty_1 and ty_2 are their corresponding types. Similarly, designate the final stack as S_{final} = (Pair $opl\ stg$, pair (operation list) ty_2) :: $[\,]$, where opl repesents a operation list.

2.2 Rules

Each instruction is defined by a rule on the system model. The rule semantic is defined by several kinds of transitions:

1. $\longrightarrow_E$ single-step evaluation of an expression in a system state,
2. $\longrightarrow_S$ internal transitions of a state, which is non-deterministic as shown for loop instructions, such as LOOP and LOOP-LEFT,
3. $\longrightarrow$ symbolic system transitions.

During the symbolic execution, numerous states are generated, but some may be unreachable from a symbolic initial state. This happens when the predicate P is unsatisfied, leading to the exclusion of a system state where the predicate fails.

T, U ::=
| ⟨comparable type⟩
| option ⟨type⟩
| list ⟨type⟩
| set ⟨comparable type⟩
| operation
| contract ⟨type⟩
| ticket ⟨comparable type⟩
| pair ⟨type⟩⟨type⟩
| or ⟨type⟩⟨type⟩
| lambda ⟨type⟩⟨type⟩
| map ⟨comparable type⟩⟨type⟩
| big-map ⟨comparable type⟩⟨type⟩
| bls12-381-g1
| bls12-381-g2
| bls12-381-fr
| sapling-transaction ⟨natural number constant⟩
| sapling-state ⟨natural number constant⟩
| chest
| chest-key

⟨comparable type⟩ ::=
| unit
| never
| bool
| int
| nat
| string
| chain-id
| bytes
| mutez
| key-hash
| key
| signature
| timestamp
| address
| tx-rollup-l2-address
| option ⟨comparable type⟩
| or ⟨comparable type⟩⟨comparable type⟩
| pair ⟨comparable type⟩⟨comparable type⟩ . . .

Figure 1 Types

```
                    t ::=
                        | ⟨variable⟩
                        | ⟨account constant⟩
                        | ⟨int constant⟩
                        | ⟨string constant⟩
                        | ⟨byte sequence constant⟩
                        | Unit
                        | Never
                        | True
                        | False
                        | Pair t1 t2
                        | Left t
                        | Right t
                        | Some t
                        | None
                        | {t ; ... }
                        | { Elt t1 t2 ; ... }
                        | {⟨instruction⟩; ...}
           ⟨variable⟩ ::=
                        | x
   ⟨account constant⟩ ::=
                        | balance
                        | amount
                        | sender
                        | source
                        | now
                        | level
                        | chain-id
                        | self
                        | self-address
                        | total-voting-power
                        | voting-power
        ⟨instruction⟩ ::=
                        | DROP
                        | DROP⟨natural number constant⟩
                        ...
```

Figure 2 Terms

$$\frac{\neg\ P}{\{[I,S,P]\} \cup SE \longrightarrow SE}$$

Michelson instructions are categorized based on their functions. These include such as control structures, stack manipulation, arithmetic operations, boolean operations, cryptographic operations, and operations on data structures. In our model, we classified them as one-step, multi-step, blockchain, cryptographic, branch, and loop instructions. This categorization reflects how these instructions are handled within our specific model.

2.2.1 One-step Instructions

One-step instructions have a localized effect, modifying the system state directly without branching a new state, and they end the execution after a single operation. The ABS rule exemplifies such an instruction, operating on the first element of the stack. This element must be of type int, and the instruction returns the positive value of its current value, treated as a natural number.

ABS

$$\frac{}{[(\text{ABS};I),(s_1,\text{int}) :: S,P] \longrightarrow_S [I,(x,\text{nat}) :: S,P \wedge (s_1 \geq 0 \Rightarrow x = s_1) \wedge (s_1 < 0 \Rightarrow x = -s_1)]}$$

Another example is the COMPARE instruction for natural numbers, which compares the first two elements of the stack, both having a natural number type. It returns a result of either -1, 0, or 1 based on the comparison of their values.

COMPARE

$$\frac{}{\begin{array}{c}[(\text{COMPARE};I),(s_1,\text{nat}) :: (s_2,\text{nat}) :: S,P] \longrightarrow \\ [I,(x,\text{int}) :: S,P \wedge\ (s_1 > s_2 \Leftrightarrow x = 1) \wedge\ (s_1 = s_2 \Leftrightarrow x = 0) \wedge\ (s_1 < s_2 \Leftrightarrow x = -1)]\end{array}}$$

2.2.2 Multi-step Instructions

Multi-step instructions involve the execution of a sub-sequence of instructions before returning to the main execution. The rule below models the EXEC instruction, which executes a function specified as a sequence of instructions denoted by I_1 in the rule. This instruction applies the code of the function to the first element of the stack and then places the result back into the main execution.

EXEC

$$\frac{[I_1,(s_1,ty_1) :: [\,],Q] \longrightarrow_S^* [[\,],(s_1',ty_2) :: Q']}{[(\text{EXEC};I),(\{I_1\},ty_1\ \rightarrow\ ty_2) :: (s_1,ty_1) :: S,P \wedge Q] \longrightarrow_S\ [I,(s_1',ty_2) :: S,P \wedge Q']}$$

The DIP n instruction delves into the first n elements of a stack, applies the sequence of instructions denoted as I_1 to the remaining part of the stack, modifies a part of the predicate and subsequently pushes the result back into the main execution.

DIP n

$$\frac{\text{len}(A)=\ n \qquad [I_1,B,Q] \longrightarrow_S^* [[\,],B_1,Q']}{[(\text{DIP}\ n\ I_1;I),A\ @\ B,P \wedge Q] \longrightarrow_S [(I),A\ @\ B_1,P \wedge Q']}$$

2.2.3 Blockchain Instructions

Blockchain instructions make implicit use of data from an execution environment, derived from the current state of the blockchain (the context), from the transaction that triggered the contract, and from the block containing this transaction. We model these data as symbolic values, which satisfis some constrants. For example, the AMOUNT instruction adds to the stack the amount of tokens that sent together with the transcation which calls this smart contract. Here, the amount is modeled as a symbolic value donated as amount, which has mutez type, which has a nature number value.

$$\textsc{AMOUNT}\quad \frac{}{[(\text{AMOUNT}; I), S, P] \longrightarrow_S [I, (\text{amount}, \text{mutez}) :: S, P}$$

2.2.4 Cryptographic Instructions

Cryptographic instructions in Michelson utilize built-in functions that manage cryptographic features. These functions are symbolically modeled in our framework, specifying their inputs and outputs. For instance, the instruction SHA256 computes the cryptographic hash of the top of the stack using the SHA-256 cryptographic hash function. The function SHA-256 is symbolically represented as the function sha256 that takes an input of type byte and returns another byte value, constrained to a length of 256.

$$\textsc{SHA256}\quad \frac{}{[(\text{SHA256}; I), (s_1, \text{byte}) :: S, P] \longrightarrow_S [I, (x, \text{byte}) :: S, P \wedge (x = \text{sha256}(s_1))]}$$

2.2.5 Branch Instructions

The Michelson language incorporates conditional branching, enabling the execution of specific code segments based on certain criteria. One of the instructions facilitating this is the IF {} {} construct. For example, the IF { I_1 } { I_2 } instruction allows the creation of branches in execution. It takes two sequences as arguments. Expecting a boolean value at the top of the stack, it consumes this top element. If the boolean is True, it executes the first provided sequence; otherwise, it executes the second sequence.

$$\textsc{IF-true}\quad \frac{}{[(\text{IF } I_1\ I_2; I), (s_1, \text{bool}) :: S, P] \longrightarrow_S [I_1, S, P \wedge s_1]}$$

$$\textsc{IF-false}\quad \frac{}{[(\text{IF } I_1\ I_2; I), (s_1, \text{bool}) :: S, P] \longrightarrow_S [I_2, S, P \wedge \neg s_1]}$$

Another branch instruction is the IF-LEFT instruction, which is used for conditional execution based on the left or right branch of a tagged union (sum) type.

$$\textsc{IF-LEFT-left}\quad \frac{}{[(\text{IF-LEFT } I_1\ I_2; I), (s_1, \text{or } ty_1\ ty_2) :: S, P] \longrightarrow_S [I_1, (x, ty_1) :: S, P \wedge (s_1 = \text{Left } x)]}$$

$$\textsc{IF-LEFT-right}\quad \frac{}{[(\text{IF-LEFT } I_1\ I_2; I), (s_1, \text{or } ty_1\ ty_2) :: S, P] \longrightarrow_S [I_2, (x, ty_2) :: S, P \wedge (s_1 = \text{Right } x))]}$$

These above rules collectively define the behavior of the IF-LEFT instruction based on the left or right branch of a tagged union type. This rule applies when the top element of the stack is a tagged union of type or ty_1 ty_2(either of type ty_1 or ty_2), and the IF-LEFT instruction is followed by instructions I_1 and I_2. The first rule apply with the condition that the top element of the stack is of the form Left x (indicating the left branch of the union), it transitions to execute the instructions I_1 with the stack modified accordingly and the predicate is updated to include the condition that the top element of the stack is equal to Left x. Similerly, the second rule apply with the condition that the top element is of the form Right x (indicating the right branch of the union), it transitions to execute the instructions I_2 with the stack modified accordingly and the predicate is updated to include the condition that the top element of the stack is equal to Right x.

2.2.6 Loop Instructions

In Michelson, loops are fundamental constructs that repeatedly execute a set of instructions until a specific condition is met. There are various loop instructions, including ITER, MAP, CONCAT, SIZE, LOOP, and LOOP-LEFT, enable the manipulation and traversal of different data structures like lists, sets, maps, and the stack itself.

Michelson, similar to other programming languages, classifies loop instructions into two main types:

- *For* loops: These are sequence-based loops, exemplified by ITER, MAP, CONCAT and SIZE.
- *While* loops: These are predicates-based loops, illustrated by LOOP and LOOP-LEFT.

To illustrate the structures of loops, we can draw a parallel with OCaml syntax:

```
while <loop_condition> do
   <loop_body>
done
```

This *While* loop structure involves a loop condition that is evaluated at the beginning of each iteration. If the condition holds true, the loop body is executed, and the process repeats until the condition becomes false. In Michelson, this pattern is evident in instructions like LOOP and LOOP-LEFT, which rely on predicate-based conditions for their iteration.

```
LOOP {<loop-body>}
```

Additionally, Michelson features a *For* loop, where the index runs from the initial value to the final value:

```
for index = <initial_value> to <final_value> do
  <loop_body>
done
```

This is represented by the ITER instruction:

```
ITER {<loop-body>}
```

The fundamental concept underlying the definition of a loop is its loop condition, the predicate that must hold true for the loop to continue execution. In the case of *While* loops, the loop condition is explicitly embedded in the loop syntax. However, for the *For* loops, the loop condition is more implicit. A *For* loop can be rewritten as a *While* loop to make the loop condition explicit:

```
index = 0
while index < length(sequence) do
  <loop_body>
  index += 1
done
```

Here, the implicit loop condition asserts that the index of the sequence element is less than the length of the sequence.

It's important to note that, unlike imperative-style loops, Michelson loop instructions do not explicitly contain their loop conditions. Instead, the loop condition is established:

- In the expressions preceding the loop for the first time entering the loop.
- Inside the loop body for subsequent iterations.

The loop instructions are applied to a stack, and the loop condition is checked with the top element of the stack. For a *While* loop the top element of the stack must have a boolean value and for the *For* loop, it must have a sequence data type, such as list, set or map. If we denote the function to retrieve the top element of a stack as "pop," the *While* loop instruction can be expressed as:

```
while pop(S) do
   loop_body
done
```

and the *For* loop is expressed as:

```
index = 0
while index < length(pop(S)) do
  <loop_body>
  index += 1
done
```

Certainly, let's continue exploring how these loop constructs are modeled in our symbolic execution.

2.2.6.1 ITER

Lets us consider the ITER instruction, which iterates over the list data structure. The corresponding typing rule is as follows:

$$\frac{\Gamma \vdash I : ty : A \ \rightarrow \ A}{\Gamma \vdash \text{ITER } I : \text{list } ty : A \ \rightarrow \ A}$$

This typing rule ensures that if the loop body I has the type $ty : A \rightarrow A$, then the entire ITER instruction, when applied to a stack with the top element of type ty list and the rest of the stack of type A, produces a result stack of type A. This facilitates symbolic iteration over a list while maintaining compatibility with further symbolic execution.

We delve into the symbolic execution of ITER instruction when the list is a symbolic term.

$$\frac{\text{ITER-EMPTY}}{[(\text{ITER } I_1; I), (s_1, \text{list } ty) :: S, P] \longrightarrow_S \ [I, S, P \ \wedge \ (s_1 = \{\})]}$$

This rule handles the case where the list is empty, and thus the ITER instruction does not iterate. The resulting state transitions to the next instruction with an updated stack that is actually the same as before entering the ITER intruction and an augmented predicate $P \wedge (s_1{=}\{\})$.

$$\textsc{ITER-nonempty}\ \frac{[\text{ITER}, (hd, ty) :: S, Q] \longrightarrow^*_S [[\ ], S', Q']}{\begin{array}{c}[(\text{ITER } I_1; I), (s_1, \text{list } ty) :: S, P \wedge Q] \longrightarrow_S \\ [(\text{ITER } I_1; I), \{\langle tl\rangle\}\ :: S', P \wedge Q' \wedge (s_1 = \{hd; \langle tl\rangle\})]\end{array}}$$

The next rule handles the case where the list is non-empty. It symbolically executes the instructions I_1 within the ITER block with the head of the list (hd) removed from the stack and a subpart of the predicate Q. The resulting state transitions to a new stack (S') and an updated predicate ($P \wedge$ Q' $\wedge$ ($s_1{=}\{hd;\langle tl\rangle\}$)), where $\{\langle tl\rangle\}$ represents the rest of the list. In summery, these rules capture the symbolic execution behavior of the ITER instruction, considering both the cases where the list is empty and non-empty.

When the sequence (list, set, map) is empty, the ITER loop halts. Otherwise, it continues to execute its loop body, which applies to an element of the list. If a list l is concrete, the instruction simply loops until the list ends with an empty list construction. The question arises: how do we deal with it symbolically when the list is a symbolic value? One solution could involve running the loop for a number of times to obtain the symbolic value. However, we aim for the result value to express the loop body more efficiently in a mathematical way and be capable of performing property checks.

Let again consider the instruction ITER $\{I\}$. Semantically, I denote the loop body, which is a sequence of instructions. In fact, the sequence of instructions, I, represents a lambda function lambda (ty : $A \rightarrow A$) applied to a stack of type ty :A, resulting in a stack of type A. Assuming that the first element of the stack is a list and it is a non-empty. Then the list is in the form $\{hd;\ \langle tl\rangle\}$, and the rest of the stack is $S_0 = s_1 :: s_2 :: ... :: s_n$. Then the ITER $\{I\}$ instruction apply m time of function representing I to the stack with each element hd_i of the list, where m is the length of list. The result after the fist loop is $S_1 = s_1^1 :: s_2^1 :: ... :: s_n^1$. Note that the type of the result stack must be the same as the prevoud stack. Let us assume that the element i of the stack S has the type ty_i. Let the function f_i that takes the head hd_0 of the list and all elements of the stack S_0 as s_1, s_2, ... s_n, donated as $\overline{S_0}$ and returns the i-element s_i^1 of the stack S_1. The function representing I is defined as a set of these subfunctions f_i.

$$\begin{aligned} s_1^1 &= f_1\ hd_0\ \overline{S_0} \\ s_2^1 &= f_2\ hd_0\ \overline{S_0} \\ &\cdots \\ s_n^1 &= f_n\ hd_0\ \overline{S_0} \end{aligned}$$

When the next loop is performed, the same function f_i is applied to the next element of the list and all element of the the stack S_1 and we get the s_i^2. In the same way, the the loop is performed for j time, we get the stack S_j as follows:

$$s_1^j \ = f_1 \ hd_{j-1} \ \overline{S_{j-1}}$$

$$s_2^j \ = f_2 \ hd_{j-1} \ \overline{S_{j-1}}$$

$$...$$

$$s_n^j \ = f_n \ hd_{j-1} \ \overline{S_{j-1}}$$

As result, we can reprensed the value of the stack after running the instruction ITER as the result of a fold function that apply the function f_i to the list l with the inital value is s_1, s_2, ... s_n as $\overline{S_0}$.

$$s_1^m \ = fold \ f_1 \ \overline{S_0} \ l$$

$$s_2^m \ = fold \ f_2 \ \overline{S_0} \ l$$

$$...$$

$$s_n^m \ = fold \ f_n \ \overline{S_0} \ l$$

Generally, we can represent the sequence of instructions I as the function f that takes an element of type ty of a list l and a stack S of type A and returns a stack of type A.

$$S' \ = fold \ f \ S \ l$$

Let consider an example ITER {ADD} that applies to a stack S that has a integer list as the top element and the rest $S_0 = s_1 :: s_2 :: ... :: s_n$.

$$\text{ITER } \{\text{ADD }\}/l \ :: \ S_0$$

Because of the requirement of the ADD instruction, the top element of the stack S_0, namely s_1, need to have the interger type. For each loop ADD instruction adds each element of the list to the top element of the stack and then the rest of the stack is unchanged. Let build the function f_1 with the assumtion that the list l is not empty as $\{hd; \langle tl \rangle\}$,

$$f_1 \ (hd, s_1, s_2, \ldots, s_n) = hd \ + s_1$$

$$f_2 \ (hd, s_1, s_2, \ldots, s_n) = s_2$$

$$...$$

$$f_n \ (hd, s_1, s_2, \ldots, s_n) = s_n$$

The function f_1 is the fucntion that take $(n + 1)$ intergers as inputs and result the sum of two first one

$$f_1 \ (x, x_1, x_2, \ldots, x_n) = x \ + x_1$$

since x_2, …, x_n do not paricipaint in the calculation, we can omit them as

$$f_1 \ (x, x_1) = x \ + x_1$$

The symbolic value of the top element of the stack after the ITER {ADD} loop is.

$$fold \ f_1 \ s_1 \ l$$

Because the loop does not change the rest of the stack, for all funtion f_i for the element s_i of the stack S could consider as an identy function, and then we can omit them in the implementation.

$$f_i\ (x, x_1, \ldots, x_i, \ldots, x_n) = x_i$$

The $fold$ function also can be used to symbolicly the result of other For loop intructions, such as CONCAT and SIZE.

2.2.6.2 MAP

The ITER instruction and some other can be symbolicly implemented using $fold$ function applying on the sequance data structures as list, set and map. Let consider the MAP instruction, where we can use a map function to symbolic the result of the instruction. Consider the MAP $\{i\}$ instruction for lists, which operates on a stack S, whose the top element has type ty list, and the rest has type A. The instruction returns a stack of type (ty list) : A. The corresponding typing rule is as follows:

$$\frac{\Gamma \vdash I : ty \rightarrow\ ty'}{\Gamma \vdash \text{MAP } I :\ ty \text{ list }\ : A\ \rightarrow\ ty' \text{ list }\ : A}$$

This typing rule ensures that if the loop body (I) has the type $ty \rightarrow ty$', then the entire ITER instruction produces a result stack of type ty' list : A.

MAP-EMPTY

$$\frac{}{[(\text{MAP } I_1; I), (s_1, \text{list } ty) :: S, P] \longrightarrow_S\ [I, (s_1, \text{list } ty) :: S, P\ \wedge\ (s_1 = \{\})]}$$

MAP-NONEMPTY

$$\frac{\begin{array}{c}[I_1, (hd, ty) :: [\,], Q1] \longrightarrow^*_S [[\,], (hd', ty) :: [\,], Q1'] \\ [\text{MAP } I_1, (\{<tl>\}, \text{list } ty) :: [\,], Q2] \longrightarrow^*_S [[\,], (\{<tl'>\}, \text{list } ty) :: [\,], Q2']\end{array}}{\begin{array}{c}[(\text{MAP } I_1; I), (s_1, \text{list } ty) :: S, P\ \wedge\ Q1 \wedge\ Q2] \longrightarrow_S \\ [I, (\{hd';<tl'>\}, \text{list } ty) :: S, P\ \wedge\ Q1' \wedge\ Q2' \wedge\ (s_1 = \{hd;<tl>\})]\end{array}}$$

Let consider how can we sybolicly the execution of the MAP instruction. The list of instruction I can be repeseted as a function f that takes a input of type ty and return an output of type ty'.

$$f\ \ : ty\ \rightarrow\ ty$$

The MAP instruction loops on the list l by applying the function f to each element of the list l. Symbolilly the result of the MAP $\{I\}$ to the list l is the result of a map function that apply the fucntion f that represent of the sequence of instructions i to the list.

$$\text{MAP } \{I\}\ /\ \ l\ \ :: S\ \rightarrow\ (map\ f\ l) :: S$$

Consider the example as MAP { PUSH int 1; ADD }, which applies to a stack with have a list of intergers l as the top elemnt of the stack. The function expressed as { PUSH int 1; ADD } basicly adds the number 1 to each element of the list. Let f is the function that is specified as the sequcne of instruction i. f takes an element of type int and returen another elemnt of type int

$$f\ : \text{int} \to\ \text{int}.$$

$$f\ x\ = x\ + 1$$

Then the result of the loop is a stack which has a interger list at the top whose simbolic value is *map* f ł.

2.2.6.3 LOOP

Now let consider an example of the while loop, namely the LOOP instruction that is a while loop. The LOOP { i } intruction apply to a stack S that has the top element is a boolean type. If a top element is true then and only then the loop body (i) is executed. After the body of a loop is executed then control again goes back at the top element, and checked if it is true, the same process is executed until the top element becomes false. Once the value of the top element becomes false, the control goes out of the loop. The loop condition is calucated by the loop body and the result back to the firt elevemnt of the stack.

$$\frac{\Gamma \vdash I : A \to\ \text{bool}\ : A}{\Gamma \vdash \text{MAP}\ I : \text{bool}\ : A\ \to\ A}$$

The rules for LOOP instruction.

$$\textsc{LOOP-true}\quad \frac{}{[(\text{LOOP}\ I_1; I), (s_1, \text{bool}) :: S, P] \longrightarrow_S\ [(I_1; \text{LOOP}\ I_1; I), S, P \land s_1]}$$

$$\textsc{LOOP-false}\quad \frac{}{[(\text{LOOP}\ I_1; I), (s_1, \text{bool}) ::\ S, P] \longrightarrow_S\ [I, S, P \land (\neg s_1)]}$$

let $S_0 = s_0 :: s_1 :: s_2 :: \ldots :: s_n$, where has the boolean type. Let f_i is the function that takes the top elemement of the stack and the rest of n elements of the stack, which have types ty_1, ty_2, … ty_i, … ty_n noted as $\overline{A}$ and return a single type of the element i.

$$f_0\ : \text{bool}\ \to\ \text{bool}.$$
$$\ldots$$
$$f_i\ : \text{bool}\ ty_1, ty_2, \ldots\ ty_i, \ldots\ ty_n\ \to\ ty_i.$$
$$\ldots$$

$$f_0\ s_0\ s_1\ \ldots\ s_i\ \ldots\ s_n\ = s_0^1.$$
$$\ldots$$
$$f_i\ s_0\ s_1\ \ldots\ s_i\ \ldots\ s_n\ = s_i^1.$$
$$\ldots$$

2.2.7 FAILWITH

A Michelson program can fail, employing a designated opcode. The FAILWITH instruction aborts the ongoing program and reveals the top element of the stack.

$$\frac{\text{FAILWITH}}{[(\text{FAILWITH}; I), (s_1, ty) :: S, P] \longrightarrow_S [[\,], (\text{FAIL } (s_1)) :: [\,], P]}$$

3 Domain Specific Language For Smart Contract Property Specification

Michelson, as a stack-based language, may pose a challenge for developers less familiar with it compared to more mainstream languages like Python, Java, or C. To address this, several high-level languages such as LIGO, SmartPy, and Liquidity have been created. These languages provide a more readable syntax, resembling popular programming languages, allowing developers to write Tezos smart contracts more comfortably. These high-level contracts are then compiled down to Michelson.

For Tezos users who may lack expertise in Michelson, there is a desire to verify smart contract code before interaction. Our framework addresses this by introducing a domain-specific language (DSL) that facilitates the specification of smart contract properties in a mathematical and readable format. The goal is to create a language that is accessible and easily understandable, even for users who are not Michelson experts.

To achieve this, our framework employs an SMT solver, specifically Z3, for property verification. The DSL is designed to allow users to express formulas that align with the capabilities of the SMT solver (Z3). This empowers users to write formulas that the solver can effectively handle, facilitating property verification in a more user-friendly manner.

One of the main challenges with formal verification is that it is often the requirement for users to express the system in a specialized specification language. This poses two problems: it demands expertise in formal verification from the specifier and introduces a gap between the actual code and its specification. In our framework, this gap is eliminated as the exact Michelson code used for verification precisely corresponds to the actual code. Users are not obligated to possess formal verification expertise.

The Michelson code, readily available on the Tezos blockchain, remains consistent across various high-level language platforms like LIGO, SmartPy, and Liquidity, all of which offer functions to compile programs into Michelson. This approach eliminates the need for users to be formal verification experts. The essential Michelson knowledge revolves around understanding data types and their usage when interacting with a smart contract on the Tezos blockchain, whether as parameters or in storage.

Given a smart contract code c, the syntax of our property specification language is as follows:

```
MContract ‘Name’ with
Code := {c}
Input := t
Output := t’
Pre-condition := cd
Post-condition := cd’
```

Here, $mcontract$, $with$, $code$, $input$, $output$, $pre-condition$, and $post-condition$are keywords. The ’Name’ is replaced by the contract name. In the input and output fields, the input and output patterns are specified. Since Michelson operates on a stack, we assume that the stack has only one element, as specified in the input pattern. Similarly, the output stack is assumed to be a stack with only one element, which is specified in the output field. As we have mentioned before, the output should be a pair consisting of an operation list and a storage. In our syntax, for verification purpose, we omit the operation list and then the output is represented by the patern of the value in the storage after the execution is finished.

The input and output patterns represent Michelson terms, which can be either concrete values or symbolic values, including variables. When a variable appears for the first time, it needs to be accompanied by its corresponding type declaration. However, subsequent mentions can omit the type declaration. Formally, the pre and post-condition instances are formulas in first-order logic, and they can be handled by an SMT solver, namely the Z3 solver in our framework. The detail of the language grammar is described in in Figure **??**.

Let's consider the following example of the property specification for the contract named Add:

```
MContract ‘Add’ with
Code := {UNPAIR; COMPARE; GE;
        IF {ADD}{PUSH string ’Unexpected pair’; FAILWITH}}
Input := Pair (x : int) (y : int)
Output := (z : int)
Pre-condition := (x >= 5) | (y >= 3)
Post-condition := (z >= 8)
```

Here, the program takes as input a stack with only one element, which is a pair of two integers, symbolically specified in the Input field. It then unpairs the input pair to obtain two elements as two integers x and y. It then proceeds to destructure the input pair, yielding two integers. Subsequently, the program performs a comparison between these two integers. If x is greater than or equal to y, the program constructs a new stack with a solitary element at the top representing the sum of x and y, designated as z. Otherwise, the execution terminates with an error message containing the content 'Unexpected pair.' The specification continues by defining pre and post conditions, where the precondition states the assumption that the integer x is greater than or equal to 5 and y is greater than or equal to 3. The postcondition asserts that the result sum z is greater than or equal to 8.

A smart contract may offer various functions for users to interact. Therefore, our language is designed to allow users to specify different functions, referred to as entrypoints, in Michelson. This design enables us to specify and verify all functions within a smart contract. Furthermore, this design allows us to describe and verify relationships between functions. The syntax is as follows:

```
mcontract Name = spec
   entrypoint %a
      code := {c1}
      input := t1
      output := t1’
      pre-condition := cd1
      post-condition := cd1’
   entrypoint %b
      code := {c2}
      input := t2
      output := t2’
      pre-condition := cd2
      post-condition := cd2’
```

```
  entrypoint %c
     ...
```

To illustrate this, let's consider the following auction smart contract. This contract has two entrypoints, bid and close, which serve as functions for bidding and closing the contract, respectively. The smart contract storage on the blockchain is a pair containing the first element as a boolean value that indicates whether the contract is still open for bidding or already closed. The second element is another nested pair, where the first element stores the highest bidder's address, and the second element is the contract owner's address.

Closing the contract transfers the balance to the owner and is restricted to the owner. Both closing and bidding operations fail if the auction is already closed. If bidding is open and the amount of tokens accompanying the bid exceeds the current highest bid, the current bidder replaces the previous highest bidder, and the previous highest bidder is reimbursed. Otherwise, bidding fails.

Upon deployment, the owner deposits an initial balance to indicate the minimum bid. The contract is supposed to be deployed with the storage value Pair True (Pair owner owner), indicating that bidding is allowed, and the contract owner is currently the highest bidder.

```
mcontract Auction = spec
  entrypoint %bid
    code := {...}
    input := Pair Unit (Pair (auction_open: bool)
                             (Pair (highest_bid: mutez)
                                   (contract_owner: address)))
    output :=   Pair (new_contract_open: bool)
                    Pair (new_highest_bidder: address)
                         (new_contract_owner: address))
    pre-condition := (auction_open = true) |
                     (Amount > Balance)
    post-condition := (Post_Balance = Amount) |
                      (new_highest_bidder = Sender) |
                      (new_contract_owner = contract_owner) |
                      Transfer_token (Unit, Balance, highest_bidder) ;
  entrypoint %close
    code := {...}
    input := Pair Unit (Pair (auction_open: bool)
                             (Pair (highest_bid: mutez)
                                   (contract_owner: address)))
    output :=   Pair (new_contract_open: bool)
                    Pair (new_highest_bidder: address)
                         (new_contract_owner: address))
    pre-condition := (auction_open = true) |
                     (Sender = contract_owner)
    post-condition := (Post_Balance = 0) |
                      Transfer_token (Unit, Balance, contract_owner) ;
    (%create -> %close) with (auction_open = true)  && (Amount > Balance) |
    (%create  -> %close) with (auction_open = true) && (Sender > contract_owner) |
    (%bid -> %bid) with (auction_open = true) && (Amount > Balance)  |
    (%bid -> %close) with (auction_open = true) && (Sender > Balance)  |
    not (%close -> %bid) |
    not (%close -> %close)"
```

4 Static Checker

Formulas that are used in assertions or added to solvers are terms of Boolean sort. Otherwise, terms of Boolean and non-Boolean sort may be mixed in any combination where sorts match up. Universal and existential quantifiers bind variables to the scope of the quantified formula.

5 Case Studies

5.1 USDtz

5.2 Kolibri Oracle Contract

6 Related Work

7 Conclusion

8 Appendix

8.0.1 Basic Instructions

8.0.1.1 Control structures

EXEC

$$\frac{[I_1, (s_1, ty_1) :: [\,], Q] \longrightarrow_S^* [[\,], (s_1', ty_2) :: Q']}{[(\text{EXEC}; I), (\{I_1\}, ty_1 \ \to\ ty_2) :: (s_1, ty_1) :: S, P \wedge Q] \longrightarrow_S \ [I, (s_1', ty_2) :: S, P \wedge Q']}$$

APPLY

$$\frac{}{\begin{gathered}[(\text{APPLY}; I), (s_1, ty_1) :: (\{I_1\}, \text{lambda (pair } ty_1\ ty_2)\ ty_3) :: S, P] \longrightarrow_S \\ [I, (\{\text{PUSH } ty_1\ s_1; \text{PAIR}; I_1\}, \text{lambda } ty_2\ ty_3) :: S, P]\end{gathered}}$$

LAMBDA

$$\frac{}{[(\text{LAMBDA } ty_1\ ty_2\ \{I_1\}; I), S, P] \longrightarrow_S \ [I, (\{I_1\}, \text{lambda } ty_1 \ \to\ ty_2) :: S, P]}$$

8.0.1.2 Stack Manipulation

DIG

$$\frac{\text{len}(A) = n}{[(\text{DIG } n; I), A \ @\ (s_1, ty) :: B, P] \longrightarrow_S [I, (s_1, ty) :: A \ @\ B, P]}$$

DIP

$$\frac{[I_1, S, Q] \longrightarrow_S^* [[\,], S_1, Q']}{[(\text{DIP } I_1; I), (s_1, ty) :: S, P \wedge Q] \longrightarrow_S [I, (s_1, ty) :: S_1, P \wedge Q']}$$

DIP N

$$\frac{\text{len}(A) = n \qquad [I_1, B, Q] \longrightarrow_S^* [[\,], B_1, Q']}{[(\text{DIP } n\ I_1; I), A \ @\ B, P \wedge Q] \longrightarrow_S [(I), A \ @\ B_1, P \wedge Q']}$$

$$
\begin{array}{rl}
T, U ::= & \\
 & \mid \langle\text{comparable type}\rangle \\
 & \mid \text{option } \langle\text{type}\rangle \\
 & \mid \text{list } \langle\text{type}\rangle \\
 & \mid \text{set } \langle\text{comparable type}\rangle \\
 & \mid \text{operation} \\
 & \mid \text{contract } \langle\text{type}\rangle \\
 & \mid \text{ticket } \langle\text{comparable type}\rangle \\
 & \mid \text{pair } \langle\text{type}\rangle\langle\text{type}\rangle \\
 & \mid \text{or } \langle\text{type}\rangle\langle\text{type}\rangle \\
 & \mid \text{lambda } \langle\text{type}\rangle\langle\text{type}\rangle \\
 & \mid \text{map } \langle\text{comparable type}\rangle\langle\text{type}\rangle \\
 & \mid \text{big-map } \langle\text{comparable type}\rangle\langle\text{type}\rangle \\
 & \mid \text{bls12-381-g1} \\
 & \mid \text{bls12-381-g2} \\
 & \mid \text{bls12-381-fr} \\
 & \mid \text{sapling-transaction } \langle\text{natural number constant}\rangle \\
 & \mid \text{sapling-state } \langle\text{natural number constant}\rangle \\
 & \mid \text{chest} \\
 & \mid \text{chest-key} \\
\langle\text{comparable type}\rangle ::= & \\
 & \mid \text{unit} \\
 & \mid \text{never} \\
 & \mid \text{bool} \\
 & \mid \text{int} \\
 & \mid \text{nat} \\
 & \mid \text{string} \\
 & \mid \text{chain-id} \\
 & \mid \text{bytes} \\
 & \mid \text{mutez} \\
 & \mid \text{key-hash} \\
 & \mid \text{key} \\
 & \mid \text{signature} \\
 & \mid \text{timestamp} \\
 & \mid \text{address} \\
 & \mid \text{tx-rollup-l2-address} \\
 & \mid \text{option } \langle\text{comparable type}\rangle \\
 & \mid \text{or } \langle\text{comparable type}\rangle\langle\text{comparable type}\rangle \\
 & \mid \text{pair } \langle\text{comparable type}\rangle\langle\text{comparable type}\rangle \ldots
\end{array}
$$

Figure 3 Types

PUSH
$$\frac{}{[(\text{PUSH } ty\ x; I), S, P] \longrightarrow_S [I, (x, ty) :: S, P]}$$

8.0.1.3 Arithmetic operations

ADD
$$\frac{}{[(\text{ADD}; I), (s_1, \text{nat}) :: (s_2, \text{nat}) :: S, P] \longrightarrow_S [I, (x, \text{nat}) :: S, P \wedge (x = s_1 + s_2)]}$$

ABS
$$\frac{}{[(\text{ABS}; I), (s_1, \text{int}) :: S, P] \longrightarrow_S [I, (x, \text{nat}) :: S, P \wedge (s_1 \geq 0 \Rightarrow x = s_1) \wedge (s_1 < 0 \Rightarrow x = -s_1)]}$$

COMPARE-NAT
$$\frac{}{\begin{array}{c}[(\text{COMPARE}; I), (s_1, \text{nat}) :: (s_2, \text{nat}) :: S, P] \longrightarrow \\ [I, (x, \text{int}) :: S, P \wedge (s_1 > s_2 \Leftrightarrow x = 1) \wedge (s_1 = s_2 \Leftrightarrow x = 0) \wedge (s_1 < s_2 \Leftrightarrow x = -1)]\end{array}}$$

COMPARE-SOME-SOME
$$\frac{[\text{COMPARE}, (x, ty) :: (y, ty) :: [\ ], Q] \longrightarrow_S^* [[\ ], (a, \text{int}) :: [\ ], Q']}{\begin{array}{c}[(\text{COMPARE}; I), (s_1, \text{option } ty) :: (s_2, \text{option } ty) :: S, P \wedge Q] \longrightarrow \\ [I, (a, \text{int}) :: S, P \wedge (s_1 = \text{Some } x) \wedge (s_2 = \text{Some } y) \wedge Q']\end{array}}$$

COMPARE-SOME-NONE
$$\frac{}{\begin{array}{c}[(\text{COMPARE}; I), (s_1, \text{option } ty) :: (s_2, \text{option } ty) :: S, P] \longrightarrow \\ [I, (1, \text{int}) :: S, P \wedge (s_1 = \text{Some } x) \wedge (s_2 = \text{None})]\end{array}}$$

8.0.1.4 Boolean operations

XOR
$$\frac{}{[(\text{XOR}; I), (s_1, \text{bool}) :: (s_2, \text{bool}) :: S, P] \longrightarrow_S [I, (x, \text{bool}) :: S, P \wedge (x = s_1 \text{ xor } s_2)]}$$

8.0.1.5 Crytographic oprerations

HASH-KEY
$$\frac{}{[(\text{HASH-KEY}; I), (s_1, \text{byte}) :: S, P] \longrightarrow_S [I, (x, \text{byte}) :: S, P \wedge (x = \text{hash-key}(s_1))]}$$

8.0.1.6 Blockchain operations

AMOUNT
$$\frac{}{[(\text{AMOUNT}; I), S, P] \longrightarrow_S [I, (\text{amount}, \text{mutez}) :: S, P}$$

CONTRACT TY - SOME
$$\frac{}{\begin{array}{c}[(\text{CONTRACT } ty; I), (s_1, \text{address}) :: S, P] \longrightarrow [I, (\text{Some } x, \text{option (contract } ty)) :: S, \\ P \wedge (\text{get-contract-type}(s_1, ty) = \text{Some } x)]\end{array}}$$

CONTRACT TY - NONE
$$\frac{}{[(\text{CONTRACT } ty; I), (s_1, \text{address}) :: S, P] \longrightarrow [I, \text{None} :: S, P \wedge (\text{get-contract-type}(s_1, ty) = \text{None}]}$$

358 #### 8.0.1.7 Operations on data structures

CAR

$$\frac{}{[(\text{CAR}; I), (s_1, \text{pair } ty_1\ ty_2) :: S, P] \longrightarrow_S [I, (x, ty_1) :: S, P \ \wedge\ (s_1 = \text{Pair } x\ y)]}$$

359 ### 8.0.2 Branch Instructions

IF

$$\frac{}{\{[(\text{IF } I_1\ I_2; I), (s_1, \text{bool}) :: S, P]\} \cup SE \longrightarrow \{[I_1, S, P \ \wedge\ s_1]\} \cup \{[I_2, S, P \ \wedge\ \neg\ s_1]\} \cup SE}$$

The IF-LEFT I_1 I_2 instruction expects a or value at the top of the stack, it consumes this top element. If the value is Left, it executes the first provided sequence; otherwise, it executes the second sequence.

IF-LEFT

$$\frac{}{\begin{array}{c}\{[(\text{IF-LEFT } I_1\ I_2; I), (s_1, \text{or } ty_1\ ty_2) :: S, P]\} \cup SE \longrightarrow \\ \{[I_1, (x, ty_1) :: S, P \wedge (s_1 = \text{Left } x)]\} \cup \{[I_2, (x, ty_2) :: S, P \wedge (s_1 = \text{Right } x))]\} \cup SE\end{array}}$$

IF-CONS-EMPTY

$$\frac{}{[(\text{IF-CONS } I_1\ I_2; I), (s_1, \text{list } ty) :: S, P] \longrightarrow_S [I_2; I, S, P \ \wedge\ (s_1 = \{\})]}$$

IF-CONS-NONEMPTY

$$\frac{}{\begin{array}{c}[(\text{IF-CONS } I_1\ I_2; I), (s_1, \text{list } ty) :: S, P], SE \ \longrightarrow_S \\ [I_1, (hd, ty) :: (\{<tl>\}, \text{list } ty) :: S, P \ \wedge\ (s_1 = \{hd; \langle tl \rangle\})]\end{array}}$$

IF-NONE-NONE

$$\frac{}{[(\text{IF-NONE } I_1\ I_2; I), (s_1, \text{option } ty) :: S, P], SE \ \longrightarrow_S [I_1; I, S, P \ \wedge\ (s_1 = \text{None})]}$$

IF-NONE-SOME

$$\frac{}{[(\text{IF-NONE } I_1\ I_2; I), (s_1, \text{option } ty) :: S, P], SE \ \longrightarrow_S [I_2, (x, ty) :: S, P \ \wedge\ (s_1 = \text{Some } x)]}$$

360 ### 8.0.3 Loop Instructions

361 #### 8.0.3.1 While loops

LOOP-TRUE

$$\frac{}{[(\text{LOOP } I_1; I), (s_1, \text{bool}) :: S, P] \longrightarrow_S [(I_1; \text{LOOP } I_1; I), S, P \wedge s_1]}$$

LOOP-FALSE

$$\frac{}{[(\text{LOOP } I_1; I), (s_1, \text{bool}) :: \ S, P] \longrightarrow_S [I, S, P \wedge (\neg s_1)]}$$

362 #### 8.0.3.2 For loops

ITER-EMPTY

$$\frac{}{[(\text{ITER } I_1; I), (s_1, \text{list } ty) :: S, P] \longrightarrow_S [I, S, P \ \wedge\ (s_1 = \{\})]}$$

ITER-<span style="font-variant:small-caps">nonempty</span>

$$\frac{[\text{ITER}, (hd, ty) :: S, Q] \longrightarrow^*_S [[\,], S', Q']}{\begin{array}{c}[(\text{ITER } I_1; I), (s_1, \text{list } ty) :: S, P \wedge Q] \longrightarrow_S \\ [(\text{ITER } I_1; I), \{\langle tl\rangle\} \; :: S', P \wedge Q' \wedge (s_1 = \{hd; \langle tl\rangle\})]\end{array}}$$

CONCAT

$$\frac{}{[(\text{CONCAT}; I), (s_1, \text{list string}) :: S, P] \longrightarrow_S [I, (\text{””}, \text{string}) :: S, P \;\wedge\; (s_1 = \{\})]}$$

CONCAT

$$\frac{[\text{CONCAT}, (\{<tl>\}, \text{list string}) :: [\,], Q] \longrightarrow^*_S [[\,], (s_2, \text{string}) :: [\,], Q']}{\begin{array}{c}[(\text{CONCAT}; I), (s_1, \text{list string}) :: S, P \;\wedge\; Q] \longrightarrow_S \\ [I, (hd\hat{}\,s_2, \text{string}) :: S, P \;\wedge\; (s_1 = \{hd;<tl>\}) \wedge\; Q']\end{array}}$$

MEM-<span style="font-variant:small-caps">empty</span>

$$\frac{}{[(\text{MEM}; I), (s_1, ty_1) :: (s_2, \text{map } ty_1 \; ty_2) :: S, P] \longrightarrow_S \; [I, (\text{False}, \text{bool}) :: S, P \;\wedge\; (s_2 = \{\})]}$$

MEM-<span style="font-variant:small-caps">nonempty</span>

$$\frac{[\text{COMPARE}, (s_1, ty_1) :: (k, ty_1) :: [\,], Q] \longrightarrow^*_S [[\,], (b, \text{int}) :: [\,], Q']}{\begin{array}{c}[(\text{MEM}; I), (s_1, ty_1) :: (s_2, \text{map } ty_1 \; ty_2) :: S, P \;\wedge\; Q] \longrightarrow_S \; [\text{MEM}; I, s_1 \; :: \{< m >\} :: S, \\ P \;\wedge\; Q' \wedge\; (s_2 = \{Elt\ k\ v; < m >\}) \wedge\; (b = 1)]\end{array}}$$

MEM-<span style="font-variant:small-caps">nonempty</span>

$$\frac{[\text{COMPARE}, (s_1, ty_1) :: (k, ty_1) :: [\,], Q] \longrightarrow^*_S [[\,], (b, \text{int}) :: [\,], Q']}{\begin{array}{c}[(\text{MEM}; I), (s_1, ty_1) :: (s_2, \text{map } ty_1 \; ty_2) :: S, P \;\wedge\; Q] \longrightarrow_S \\ [I, (\text{True}, \text{bool}) :: S, P \;\wedge\; Q' \wedge\; (s_2 = \{Elt\ k\ v; < m >\}) \wedge\; (b = 0)]\end{array}}$$

MEM-<span style="font-variant:small-caps">nonempty</span>

$$\frac{[\text{COMPARE}, (s_1, ty_1) :: (k, ty_1) :: [\,], Q] \longrightarrow^*_S [[\,], (b, \text{int}) :: [\,], Q']}{\begin{array}{c}[(\text{MEM}; I), (s_1, ty_1) :: (s_2, \text{map } ty_1 \; ty_2) :: S, P \;\wedge\; Q] \longrightarrow_S \\ [I, (\text{False}, \text{bool}) :: S, P \;\wedge\; Q' \wedge\; (s_2 = \{Elt\ k\ v; < m >\}) \wedge\; (b = -1)]\end{array}}$$

MAP-<span style="font-variant:small-caps">empty</span>

$$\frac{}{[(\text{MAP } I_1; I), (s_1, \text{list } ty) :: S, P] \longrightarrow_S \; [I, (s_1, \text{list } ty) :: S, P \;\wedge\; (s_1 = \{\})]}$$

MAP-<span style="font-variant:small-caps">nonempty</span>

$$\frac{\begin{array}{c}[I_1, (hd, ty) :: [\,], Q1] \longrightarrow^*_S [[\,], (hd', ty) :: [\,], Q1'] \\ [\text{MAP } I_1, (\{<tl>\}, \text{list } ty) :: [\,], Q2] \longrightarrow^*_S [[\,], (\{<tl'>\}, \text{list } ty) :: [\,], Q2']\end{array}}{\begin{array}{c}[(\text{MAP } I_1; I), (s_1, \text{list } ty) :: S, P \;\wedge\; Q1 \wedge\; Q2] \longrightarrow_S \\ [I, (\{hd';<tl'>\}, \text{list } ty) :: S, P \;\wedge\; Q1' \wedge\; Q2' \wedge\; (s_1 = \{hd;<tl>\})]\end{array}}$$

UPDATE-<span style="font-variant:small-caps">empty-true</span>

$$\frac{}{\begin{array}{c}[(\text{UPDATE}; I), (x, ty) :: (b, \text{bool}) :: (s_1, \text{list } ty) :: S, P] \;\longrightarrow \\ [I, (\{x\}, \text{list } ty) :: S, P \;\wedge\; (s_1 = \{\}) \wedge\; (b = \text{True})]\end{array}}$$

$$\textsc{UPDATE-empty-false}\quad \frac{}{\begin{array}{c}[(\text{UPDATE}; I), (x, ty) :: (b, \text{bool}) :: (s_1, \text{list } ty) :: S, P] \longrightarrow \\ [I, (\{\}, \text{list } ty) :: S, P \wedge (s_1 = \{\}) \wedge (b = \text{False})]\end{array}}$$

$$\textsc{UPDATE-nonempty-true}\quad \frac{[\text{COMPARE}, (x, ty) :: (hd, ty) :: [\,], Q] \longrightarrow^*_S [[\,], (a, \text{int}) :: [\,], Q']}{\begin{array}{c}[(\text{UPDATE}; I), (x, ty) :: (b, \text{bool}) :: (s_1, \text{list } ty) :: S, P \wedge Q] \longrightarrow [I, (s_1, \text{list } ty) :: S, \\ P \wedge Q' \wedge (s_1 = \{hd;<tl>\}) \wedge (b = \text{True}) \wedge (a = 0)]\end{array}}$$

$$\textsc{UPDATE-nonempty-false}\quad \frac{[\text{COMPARE}, (x, ty) :: (hd, ty) :: [\,], Q] \longrightarrow^*_S [[\,], (a, \text{int}) :: [\,], Q']}{\begin{array}{c}[(\text{UPDATE}; I), (x, ty) :: (b, \text{bool}) :: (s_1, \text{list } ty) :: S, P \wedge Q] \longrightarrow \\ [I, (\{<tl>\}, \text{list } ty) :: S, \quad P \wedge Q' \wedge (s_1 = \{hd;<tl>\}) \wedge (b = \text{False}) \wedge (a = 0)]\end{array}}$$

$$\textsc{UPDATE-nonempty-true}\quad \frac{[\text{COMPARE}, (x, ty) :: (hd, ty) :: [\,], Q] \longrightarrow^*_S [[\,], (a, \text{int}) :: [\,], Q']}{\begin{array}{c}[(\text{UPDATE}; I), (x, ty) :: (b, \text{bool}) :: (s_1, \text{list } ty) :: S, P \wedge Q] \longrightarrow \\ [I, (\{x; hd;<tl>\}, \text{list } ty) :: S, \\ P \wedge Q' \wedge (s_1 = \{hd;<tl>\}) \wedge (b = \text{True}) \wedge (a = -1)]\end{array}}$$

$$\textsc{UPDATE-nonempty-false}\quad \frac{[\text{COMPARE}, (x, ty) :: (hd, ty) :: [\,], Q] \longrightarrow^*_S [[\,], (a, \text{int}) :: [\,], Q']}{\begin{array}{c}[(\text{UPDATE}; I), (x, ty) :: (b, \text{bool}) :: (s_1, \text{list } ty) :: S, P \wedge Q] \longrightarrow \\ [I, (\{hd;<tl>\}, \text{list } ty) :: S, \quad P \wedge Q' \wedge (s_1 = \{hd;<tl>\}) \wedge (b = \text{False}) \wedge (a = -1)]\end{array}}$$

$$\textsc{UPDATE-nonempty-1}\quad \frac{\begin{array}{c}[\text{COMPARE}, (x, ty) :: (hd, ty) :: [\,], Q1] \longrightarrow^*_S [[\,], (a, \text{int}) :: [\,], Q1'] \\ [\text{UPDATE}, (x, ty) :: (b, \text{bool}) :: (\{<tl>\}, \text{list } ty) :: [\,], Q2] \longrightarrow^*_S \\ [[\,], (\{<tl'>\}, \text{list } ty) :: [\,], Q2']\end{array}}{\begin{array}{c}[(\text{UPDATE}; I), (x, ty) :: (b, \text{bool}) :: (s_1, \text{list } ty) :: S, P \wedge Q1 \wedge Q2] \longrightarrow \\ [I; (\{hd;<tl'>\}, \text{list } ty) :: S, P \wedge Q1' \wedge Q2' \wedge (s_1 = \{hd;<tl>\}) \wedge (a = 1)]\end{array}}$$

363 ### 8.0.4 Operation on tickets

364 ### 8.0.5 FAILWITH

365 a Michelson program can fail, explicitly using a specific opcode

$$\textsc{FAILWITH}\quad \frac{}{[(\text{FAILWITH}; I), (s_1, ty) :: S, P] \longrightarrow_S [[\,], \mathit{failwith}(s_1) :: [\,], P]}$$

8.1 Dealing with loops

8.1.1 For loop: ITER

Semantic

$$\frac{}{\text{ITER } I \ /\{\} :: \ S \ \rightarrow \ S}$$

$$\frac{I \ /x \ :: \ S \rightarrow \ S'}{\text{ITER } I \ /\{x \ ;\text{<}tl\text{>} \ \} :: \ S \ \rightarrow \ \text{ITER } I \ /\{\text{<}tl\text{>} \ \} :: \ S'}$$

8.1.2 While loops: LOOP and LOOP-LEFT

```
                            t ::=
                              | ⟨variable⟩
                              | ⟨account constant⟩
                              | ⟨int constant⟩
                              | ⟨string constant⟩
                              | ⟨byte sequence constant⟩
                              | Unit
                              | True
                              | False
                              | Pair t1 t2
                              | Left t
                              | Right t
                              | Some t
                              | None
                              | {t ; ... }
                              | { Elt t1 t2 ; ... }
                              | {⟨instruction⟩; ...}
                   ⟨variable⟩ ::=
                              | x
           ⟨account constant⟩ ::=
                              | balance
                              | amount
                              | sender
                              | source
                              | now
                              | level
                              | chain-id
                              | self
                              | self-address
                              | total-voting-power
                              | voting-power
    ⟨natural number constant⟩ ::=
                              | [0-9]+
               ⟨int constant⟩ ::=
                              | ⟨natural number constant⟩
                              | -⟨natural number constant⟩
            ⟨string constant⟩ ::=
                              | "⟨string content⟩*"
                ⟨instruction⟩ ::=
                              | DROP
                              | DROP⟨natural number constant⟩
                              ...
```

Figure 4 Terms

$$
\begin{array}{rl}
\text{p} & ::= \\
 & \mid\ \langle\text{atomic formula}\rangle \\
 & \mid\ \neg\,\text{p}\ \mid\ \text{p}\ \wedge\ \text{q}\ \mid\ \text{p}\ \vee\ \text{q} \\
\langle\text{atomic formula}\rangle & ::= \\
 & \mid\ \langle\text{butop}\rangle\langle\text{bterm}\rangle \\
 & \mid\ \langle\text{bterm}\rangle\langle\text{biop}\rangle\langle\text{bterm}\rangle \\
\langle\text{butop}\rangle & ::= \\
 & \mid\ \text{not} \\
\langle\text{biop}\rangle & ::= \\
 & \mid\ =\ \mid\ >\ \mid\ <\ \mid\ >=\ \mid\ =<\ \mid\ != \\
 & \mid\ \text{and}\ \mid\ \text{or}\ \mid\ \text{xor} \\
\langle\text{bterm}\rangle & ::= \\
 & \mid\ \text{t} \\
 & \mid\ \langle\text{unop}\rangle\text{t} \\
 & \mid\ \text{t}\ \langle\text{binop}\rangle\text{t} \\
\langle\text{unop}\rangle & ::= \\
 & \mid\ \text{abs} \\
 & \mid\ \text{size} \\
 & \mid\ \text{int} \\
 & \mid\ \text{contract}\langle\text{type}\rangle \\
 & \mid\ \text{isleft} \\
 & \mid\ \text{isnone} \\
 & \mid\ \text{neg} \\
 & \mid\ \text{blake2b} \\
 & \mid\ \text{hash-key} \\
 & \mid\ \text{keccak} \\
 & \mid\ \text{pairing-check} \\
 & \mid\ \text{sha256} \\
 & \mid\ \text{sha3} \\
 & \mid\ \text{sha512} \\
 & \mid\ \text{implicit-account} \\
 & \mid\ \text{check-sig} \\
\langle\text{bitop}\rangle & ::= \\
 & \mid\ +\ \mid\ \text{-}\ \mid\ *\ \mid\ /
\end{array}
$$

Figure 5 Predicates