A Dynamic Logic for Symbolic Execution for the Smart Contract Programming Language Michelson

Barnabas Arvay
University of Freiburg, Germany

Thi Thu Ha Doan
University of Freiburg, Germany

Peter Thiemann
University of Freiburg, Germany

Abstract

Verification of smart contracts is an important topic in the context of blockchain technology. We study an approach to verification that is based on symbolic execution.

As a formal basis for symbolic execution, we design a dynamic logic for Michelson, the smart contract language of the Tezos blockchain, and prove its soundness in the proof assistant Agda. Towards the soundness proof we formalize the concrete semantics as well as its symbolic counterpart in a unified setting. The logic encompasses single contract runs as well as inter-contract runs chained in a single transaction.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

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

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.3

Supplementary Material
Software (Source Code): `https://freidok.uni-freiburg.de/data/255176` [6]

Funding *Thi Thu Ha Doan*: Supported by the Tezos Foundation, grant COOC.

1 Introduction

Blockchain technology and smart contracts provide decentralized and immutable systems for secure transactions and automated agreements. Smart contracts have been targets of spectacular and costly attacks as contracts are immutable and their source code is publicly available on the blockchain. Hence, it is vital as well as challenging to ensure the correctness of smart contracts before their deployment. Formal methods and various verification techniques have been proposed to address this challenge.

The Tezos blockchain [14] and its smart contract language Michelson have been designed from ground up with verification in mind. Several frameworks have been developed based on, e.g., interactive theorem proving [10], refinement typing [27], and automated theorem proving [5]. We are interested in automated verification of Michelson programs, which rules out interactive approaches. Symbolic execution [20, 11] is one of the standard approaches to automatically obtain verification conditions like weakest preconditions for failures as well as normal termination from a program. Next, an SMT-solver discharges these verification conditions. There is a wide range of approaches that apply symbolic execution combined with SMT-solving to smart contracts, mostly for the Ethereum blockchain (see Section 6).

While there are many approaches to symbolic execution [12, 13, 30], we choose one based on dynamic logic. Dynamic logic (DL) [16] is a modal logic for reasoning about programs. Its signature features are modalities for program execution. These modalities enable the expression of assertions about program behavior as logical formulas. For instance, the formula

$[p]\Psi$ states partial correctness: if program p terminates, then Ψ is true. That is, a Hoare triple $\{\Phi\}\ p\ \{\Psi\}$ can be encoded by $\Phi \to [p]\Psi$. DL also provides a modality $\langle p \rangle$ for total correctness, but we do not consider it in this work.

Dynamic logic comes with proof rules for the modality derived from the structure of p. For example, if $p; q$ stands for sequential execution of p and q, then the proof rule $[p; q]\Psi \leftrightarrow [p][q]\Psi$ states that execution of p enables execution of q such that Ψ holds in the end. Similarly, the rule $[\varepsilon]\Psi \leftrightarrow \Psi$ states that the empty program ε does not modify the validity of Ψ.

In the past, dynamic logic has been used successfully for as a basis for symbolic execution in the context of the verification of Java programs [9], as it is particularly well suited to keep track of a changing environment (i.e., mutable objects on Java's heap). We design a DL to model Michelson execution because we want to reason about transactions that span several contract runs. In Michelson terminology, these transactions are called *chained contract executions*, where an externally started contract run initiates further internal contract runs. Our DL design models the relevant parts of the blockchain run-time system on top of the purely functional execution of Michelson programs. On the level of the run-time system contracts are very similar to objects: they are identified by an address and they come with mutable attributes (state and balance).

The DL treatment of the functional part of Michelson is quite intuitive: programs are sequences of Michelson instructions, we model the execution state of a Michelson program by a formula of the form $\Phi \to [p]\Psi$, and the proof rules for $[i; p]\Psi$ (where i is a single instruction) define the semantics of symbolic execution.

Gas is an important aspect of computation on the blockchain. The initial caller of a contract has to pay for executing the transaction (consisting of one or more chained contract runs) in terms of gas. A transaction that runs out of gas is rolled back by the run-time system of the blockchain as if it never happened. As Michelson does not suffer from reentrancy problems (cf. Section 2), gas does not affect reasoning about the functional correctness of (chained) contract execution. For that reason, our DL design does not account for gas.

It is the sole goal of this paper to provide a **machine verified specification** of symbolic execution for Michelson, rather than an efficient or otherwise realistic implementation. For that reason, the paper does not cover all instructions, but rather a carefully chosen representative subset. This is in contrast to related work [10, 27, 5] that describes **actual verification tools**. To be useful for a wide range of programs, such a tool must support as many Michelson instructions as possible[1], it must be reasonably efficient, and it must deal with loops and nontermination in an appropriate way. None of these issues are concerns for our specification.

Contributions

1. We select a representative subset of Michelson instructions so as to provide proof templates for all current and future instructions that work similarly.
2. We provide a parameterized semantics definition with instances for the concrete semantics as well as for an abstract semantics, which implements the dynamic logic for Michelson.
3. We prove the soundness of this logic first for single programs, and then for several programs chained in a transaction.

The Agda implementation of the contributions is available.[2]

[1] Keeping up with the rapid evolution of the language is challenging for those tools. As of this writing, most of them support the instruction set available in late 2022.

[2] `https://freidok.uni-freiburg.de/data/255176`, development version `https://github.com/Tezos-Project-Uni-Freiburg/michelson-dynamic-logic`.

Overview

Section 2 gives an overview of Michelson, introduces its type system and our intrinsically typed representation of the language. Section 3 defines the execution model of Michelson, first for single contracts, and then for the chained execution of several contracts that call each other. Section 4 introduces dynamic logic and its symbolic execution rules, again first for single execution, and then for chained execution. Section 5 explains the major components of the soundness proof of the dynamic logic. Section 6 discusses related work and conclusions.

The paper contains many excerpts from the live, type checked definitions and proofs in Agda. In particular, all major definitions and statements of theorems are shown in Agda notation to ensure consistency of the paper with the machine-checked proofs.

2 Michelson

Michelson [25, 28] is the native language for smart contracts on the Tezos blockchain. It is a low level, stack-based, simply-typed, purely functional programming language. That is, all computation is driven by transforming an input stack into an output stack. There are no mutable data structures; blockchain transactions are handled outside of Michelson. All contracts are statically typed to avoid run-time type errors.

Each Michelson instruction transforms a given input stack into an output stack where some of its values have been changed, added, or removed. For example, the `ADD` instruction accepts any stack whose two topmost elements are numbers, and returns a stack where these two values have been replaced by their sum. The remaining stack is unchanged.

$$\text{ADD} \quad : \quad 15 :: 27 :: \mathit{remainingStack} \mapsto 42 :: \mathit{remainingStack}$$

2.1 Types

Michelson supports the usual data types like numbers, pairs, and lists as well as some blockchain-specific types for tokens and contracts. Figure 1a contains Agda definitions for a select subset of Michelson types Type. As some base types can be treated alike later on, we represent them with a separate type BaseType.

Most types' names are self explanatory. The base type 'mutez stands for tokens, addr stands for blockchain addresses in Tezos. We introduce shorthand patterns for base types for readability. The type operation consists of blockchain operations that can be emitted during contract execution. This mechanism implements token transfers from the current contract to other accounts or contracts. The type contract P represents such a contract which accepts a parameter of type ty represented by P: Passable ty. The type predicate Passable : Type → Set originates from the Michelson specification and characterizes types that can be passed as parameters to contracts. Its declaration is mutually recursive with Type.

The semantics of types is defined by a mapping to Agda types. Most Michelson types have obvious Agda counterparts, except addr, contract, and operation. Addresses and contracts are both represented by natural numbers. The difference is that a value of type contract is known to be a valid address of a contract of suitable type. We only define one alternative of the Operation datatype: transfer-tokens v m c, which models a token transfer to contract c while passing the parameter value v and tokens m.[3]

[3] At the time of writing this paper, full Michelson also supports the operations `CREATE-CONTRACT`, `EMIT` (deliver an event to an external application), and `SET-DELEGATE` (delegate stakes to another account).

```
data BaseType : Set where
  'unit 'nat 'addr 'mutez : BaseType

data Type where
  operation   : Type
  base        : BaseType → Type
  pair        : Type → Type → Type
  list option : Type → Type
  contract    : ∀ {t} → Passable t → Type

pattern unit   = base 'unit
pattern nat    = base 'nat
pattern addr   = base 'addr
pattern mutez  = base 'mutez
```

(a) Syntax.

```
Addr   = ℕ - blockchain addresses
Mutez  = ℕ - Tezos currency
data Operation : Set

⟦_⟧ : Type → Set
⟦ unit ⟧        = ⊤
⟦ nat ⟧         = ℕ
⟦ addr ⟧        = Addr
⟦ mutez ⟧       = Mutez
⟦ operation ⟧   = Operation
⟦ pair t₁ t₂ ⟧  = ⟦ t₁ ⟧ × ⟦ t₂ ⟧
⟦ list t ⟧      = List ⟦ t ⟧
⟦ option t ⟧    = Maybe ⟦ t ⟧
⟦ contract P ⟧  = Addr

data Operation where
  transfer-tokens : ∀ {P : Passable t}
    → ⟦ t ⟧ → ⟦ mutez ⟧ → ⟦ contract P ⟧
    → Operation
```

(b) Semantics.

Figure 1 Michelson Types.

2.2 Programs and Instructions

Michelson programs are intrinsically typed, that is, only well-typed programs can be written. Accordingly, they are represented in Agda by a datatype Program indexed by the types on the input stack and the types on the output stack. We assume that Stack = List Type.

```
data Program : Stack → Stack → Set
data Instruction : Stack → Stack → Set

data Program where
  end : Program S S
  _;_ : Instruction Si So → Program So Se → Program Si Se
```

Instructions are indexed in the same way: If instruction `inst` maps an input stack of type Si to an output stack of type So and `prg` maps that output stack So to the final stack of type Se, then `inst ; prg` is a program that maps Si to Se. The empty program end does not transform the stack.

We discuss a representative subset of Michelson instructions shown in Figure 2. The definition of Instruction$^+$ implements the pattern that most instructions only transform a fixed number of elements on top of the input stack and are parametric in the rest.

The first group of instructions operates on a fixed number of values on the stack and pushes the result. All arithmetic operations belong to this group and we just give two examples, ADDnn and ADDm, which perform addition of natural numbers and tokens, respectively. Michelson language overloads arithmetic operators, but as overloading is not supported by Agda, we supply separate instructions. We come back to this issue at the end of this section.

```
Instruction⁺ : Stack → Stack → Set
Instruction⁺ a b = ∀ {s} → Instruction (a ++ s) (b ++ s)

data Instruction where
  ADDnn  : Instruction⁺          [ nat ; nat ]       [ nat ]
  ADDm   : Instruction⁺          [ mutez ; mutez ]   [ mutez ]
  CAR    : Instruction⁺          [ pair t₁ t₂ ]      [ t₁ ]
  CDR    : Instruction⁺          [ pair t₁ t₂ ]      [ t₂ ]
  PAIR   : Instruction⁺          [ t₁ ; t₂ ]         [ pair t₁ t₂ ]
  NONE   : ∀ t → Instruction⁺ []                     [ option t ]
  SOME   : Instruction⁺          [ t ]               [ option t ]
  NIL    : ∀ t → Instruction⁺ []                     [ list t ]
  CONS   : Instruction⁺          [ t ; list t ]      [ list t ]
  TRANSFER-TOKENS : ∀ {P : Passable t} → Instruction⁺ [ t ; mutez ; contract P ] [ operation ]

  DROP   : Instruction⁺ [ t ]          []
  DUP    : Instruction⁺ [ t ]          [ t ; t ]
  SWAP   : Instruction⁺ [ t₁ ; t₂ ]    [ t₂ ; t₁ ]
  UNPAIR : Instruction⁺ [ pair t₁ t₂ ] [ t₁ ; t₂ ]

  AMOUNT   : Instruction⁺ [] [ mutez ]
  BALANCE  : Instruction⁺ [] [ mutez ]
  CONTRACT : (P : Passable t) → Instruction⁺ [ addr ] [ option (contract P) ]

  PUSH     : Data t → Instruction⁺ [] [ t ]

  IF-NONE  : Program S Se → Program (t :: S) Se → Instruction (option t :: S) Se
  ITER     : Program (t :: S) S → Instruction (list t :: S) S
  DIP      : ∀ n → {T (n ≤ᵇ length S)} → Program (drop n S) Se → Instruction S (take n S ++ Se)
```

Figure 2 Instructions of Core Michelson.

CAR, CDR, and PAIR are the standard operations on pairs. NONE and SOME are the constructors for the option datatype, and NIL and CONS construct lists. The constructors for "empty" containers, NONE and NIL are indexed by the element type, otherwise that type can be inferred from the context.

The last instruction in this group is TRANSFER-TOKENS. Despite the name, this instruction does **not** directly transfer tokens to another account. It rather constructs a value transfer-tokens v m c of type operation from its arguments.

The instructions in the next group differ in that they push zero or more values on the output stack. DROP pops the stack, DUP duplicates the top of the stack, SWAP swaps the top entries, and UNPAIR eliminates a pair and pushes its contents. UNPAIR is a convenience instruction as it is equivalent to the instruction sequence DUP; CDR; SWAP; CAR.

The next group contains instructions that are blockchain specific. AMOUNT returns the tokens that were transferred with the currently running contract invocation and BALANCE returns the tokens currently owned by it. The CONTRACT instruction is indexed by a type t that must be Passable. It takes an address and checks on the blockchain whether this address is associated with a contract that accepts arguments of type t. The result is communicated as an option type. That is, the contract type carries a verified address.

The PUSH instruction pushes a value of type t on the stack. The value is encoded by a type-indexed datatype Data for pushable values. We elide its straightforward definition.

The last group of instructions showcases control structures and an instruction that operates in a non-uniform way on the stack. The instruction IF-NONE eliminates a value of option type from the top of the stack. Its parameters are programs that implement the branches for case None and Some. The latter takes the value wrapped in the Some constructor as an argument on top of the stack.

The instruction ITER runs a sub-program on every element of its argument list. The instruction DIP n runs a sub-program at depth n on the input stack, that is, it skips over the first n elements of the stack, runs the sub-program, and reattaches those elements. The extra machinery in the implicit argument of the instruction makes sure that there are at least n elements on the stack. This mechanism is called reflection in the PLFA textbook [33].

Earlier, we remarked that Agda does not allow overloading of constructors in the same datatype. However, we can use reflection to define a "smart constructor" that almost suits the purpose.

```
overADD : (t₁ t₂ : Type) → Maybe (∃[ t ] Instruction⁺ [ t₁ ; t₂ ] [ t ])
overADD nat    nat    = just (nat   , ADDnn)
overADD mutez  mutez  = just (mutez , ADDm)
overADD _      _      = nothing

ADD : ∀ (p : map proj₁ (overADD t₁ t₂) ≡ just t) → Instruction⁺ [ t₁ ; t₂ ] [ t ]
ADD{t₁}{t₂}{t} p with overADD t₁ t₂
... | just (t , add) with just-injective p
... | refl = add
```

The definition exploits the fact that the input stack of an instruction is always known in a Michelson program. The same fact also enables overloading in Michelson's implementation to work. The function overADD specifies the resolution of overloading for the ADD instruction. If the argument types are both nat, then the result type is nat and the chosen instruction is ADDnn; and so on.[4] If no overloading is known for a combination of arguments, the function returns nothing. The smart constructor ADD takes a proof that the overloading is defined on a given pair of input types. Then it extracts the selected instruction from the overloading.

Compared to "real" Michelson, the smart constructor requires an extra argument to work:

```
exnat : Program [ (pair nat nat) ] [ (pair nat nat) ]
exnat = DUP ; UNPAIR ; ADD refl ; DROP ; end
```

2.3 Blockchain Interface

A contract on the Tezos blockchain is indexed by a parameter type p and a store type s. The type p must be Passable and the type s must be Storable. Moreover, each contract comes with a current balance of tokens and a store of type s. The implementation of the contract is a program that maps a pair p s to a pair (list operation) s, that is, it consumes the parameter paired with the current store and produces a list of operations (e.g., to invoke further contracts) paired with the updated store. The program itself is pure; any side effects, i.e., store update and contract calls, are managed by the blockchain runtime.

[4] The full Michelson language has ten different overloadings of ADD.

```
record Contract (Mode : MODE) (p s : Type) : Set where
  constructor ctr
  field Param   : Passable p
        Store   : Storable s
        balance : M Mode mutez
        storage : M Mode s
        program : Program [ pair p s ] [ pair (list operation) s ]
```

The *Mode* argument abstracts over the semantics of types. Its type has three components, one $\mathcal{M}$ for the semantics and the others, $\mathcal{F}$ and $\mathcal{G}$, are used by the abstract semantics in Subsection 4.2.

```
record MODE : Set₁ where
  field M : Type → Set ; F : Set ; G : Set
```

Its instantiation for the concrete semantics installs the standard semantics of types from Section 2.1. The remaining components are instantiated to the unit type $\top$.

```
CMode : MODE
CMode = record { M = ⟦_⟧ ; F = ⊤ ; G = ⊤}
```

With this definition, the contract store of the blockchain is just a partial mapping from addresses to contracts.

```
Blockchain : (Mode : MODE) → Set
Blockchain Mode = Addr → Maybe (∃[ p ] ∃[ s ] Contract Mode p s)
```

To start executing a contract, we initiate a blockchain transaction to its address, i.e., we ask the blockchain runtime to transfer tokens to its address along with its parameter. Once a contract has terminated, the runtime updates the stored value and processes the list of operations.

On the Tezos blockchain a normal account with deposit *init* corresponds to a contract with a unit parameter, unit store, and a trivial program that issues no operations.

```
Account : Mutez → Contract CMode unit unit
Account init = ctr unit unit init tt (CDR ; NIL operation ; PAIR ; end)
```

3 Michelson Reference Implementation

Program execution is defined in a small-step manner by a function that maps the current execution state of a program to a new state resulting from executing the first instruction:

```
prog-step : CProgState ro → CProgState ro
```

The type CProgState *ro* is a record that contains an input stack type *ri*, a program that maps an *ri* stack to an *ro* stack, an input stack of type *ri*, and the execution environment. prog-step executes the first instruction that must map an *ri* stack to an intermediate stack of type *re*, say. Consequently, the program in the output CProgState maps an *re* stack to an *ro* stack. As instructions as well as programs are intrinsically typed, the intermediate stack type *re* is sure to match. Likewise, the typing of prog-step ensures type preservation.

```
record ProgState (Mode : MODE) (ro : Stack) : Set where
  constructor state
  field {ri} : Stack
       en  : Environment Mode
       prg : ShadowProg{ℳ Mode} ri ro
       stk : All (ℳ Mode) ri
       Φ   : ℱ Mode

prog-step ρ | fct ft ; p
  = record ρ { prg = p ; stk = app-fct ft (H.front (stk ρ)) H.++ H.rest (stk ρ) }
prog-step ρ | DROP ; p
  = record ρ { prg = p ; stk = H.rest (stk ρ) }
```

Figure 3 Program state and single program step execution (excerpt).

3.1 Program Execution

So far we only concerned ourselves with the type of a Michelson stack. For program execution, both the types and values of stack elements are relevant. To this end, we have to lift the interpretation of a single type, i.e., a function from Type to Set, to the interpretation of a list of types. The library predicate All does exactly that: it "maps" a Set-typed function over a list, which yields (the type of) a heterogeneously typed list.

For example, the value interpretation of a type stack is a value stack where corresponding elements t and v are related by the type interpretation, that is, v : ⟦t⟧.

```
Int : Stack → Set
Int = All ⟦_⟧

a-stack : Int (nat :: unit :: option addr :: [])
a-stack = 42 :: tt :: nothing :: []
```

The definition of a program state (see Figure 3) abstracts over a *Mode* which contains a type interpretation that allows us reuse the same structure for concrete execution and abstract execution. A program state contains the program that is currently executed, the stack, and an environment which provides the context information to execute blockchain instructions like AMOUNT and BALANCE. It is parameterized by the output stack type, which does not change during execution. When executing more than one contract as we demonstrate in Sec. 3.4, this parameterization ensures that the results from completed contract executions are well typed.

The function prog-step executes the first instruction of a program on the current state. We explain two exemplary cases shown in Figure 3. To explain the first stanza of the code we have to make a confession. As several instructions have very similar semantics, our internal representation of instructions is a refinement of the datatype shown in Figure 2. For example, all instructions that just apply a function to the top of the stack are grouped under a constructor fct and func-type is the type defining these instructions.

```
fct : func-type args results → Instruction⁺ args [× results ]
```

The function app-fct applies such a function to a concrete stack. Roughly speaking, if the underlying function has type $a_1 \to \cdots \to a_n \to (r_1 \times \cdots \times r_m)$ it gets transformed into a function between heterogeneously typed lists $[a_1, \ldots, a_n] \to [r_1, \ldots, r_m]$. We elide the

definition and just remark that the function [×_] implements the transformation between $(r_1 \times \cdots \times r_m)$ and $[r_1, \ldots, r_m]$. The functions H.front and H.rest (in Fig. 3) split the input stack according to the stack types expected by the function *ft*. The function H.++ is concatenation of heterogeneous lists.

The DROP instruction drops the top of the stack.

3.2 Execution of Control Flow Instructions

We have chosen a small-step semantics because its stepwise progression matches the stepwise proof rules of the dynamic logic. However, the Michelson specification defines the semantics in terms of a big-step judgment.[5]

```
record Configuration (ri : Stack) : Set where
  constructor Conf
  field cenv : CEnvironment ; stk : Int ri

data [_,_]⇓_ : Configuration ri → Program ri ro → Int ro → Set
```

It relates a configuration (environment and input stack of type *ri*) and a program to an output stack of type *ro*. The definition of the semantics in the Michelson specification takes some liberties that require some extra machinery in a small-step execution model. We discuss these issues with some representative instructions.

The instruction IF-NONE *p-none p-some* expects a value of type option on top of the stack. If that value is nothing (the encoding of NONE), the *p-none* branch is executed on the rest of the stack:

```
↓-IF-NONE : ∀ {p-none : Program txs tys} {p-some : Program (tx :: txs) tys}
  → [ Conf ce xs , p-none ]⇓ ys
  ----------------------------
  → [ Conf ce (nothing :: xs) , IF-NONE p-none p-some ]↓ ys
```

If however the top of the stack is just x (encoding SOME x), the *p-some* branch is executed on the stack where just x is replaced with x:

```
↓-IF-SOME : ∀ {p-none : Program txs tys} {p-some : Program (tx :: txs) tys}
  → [ Conf ce (x :: xs) , p-some ]⇓ ys
  ----------------------------
  → [ Conf ce (just x :: xs) , IF-NONE p-none p-some ]↓ ys
```

To specify the corresponding small-step rule we introduce a type-respecting concatenation operator ;• on programs. The program IF-NONE *p-none p-some* ; *p-rest* either transitions to *p-none* ;• *p-rest* or to *p-some* ;• *p-rest*, depending on the value on top of the stack.

The instruction DIP n p executes program p on the stack that results from removing the first n elements of the current stack and reattaches them afterwards.

```
↓-DIP : ∀ {n} {q : T (n ≤ᵇ length txs)} {p-dip : Program (drop n txs) tys}
  → [ Conf ce (H.drop n xs) , p-dip ]⇓ ys
  ----------------------------
  → [ Conf ce xs , DIP n {q} p-dip ]↓ (H.take n xs H.++ ys)
```

[5] For typing reasons the implementation splits it in four judgments for programs ⇓, instructions ↓, shadow programs ⇓, and shadow instructions $\downarrow'$.

In the small-step version, dropping the first n elements of the stack is easy, but reattaching them requires extra machinery. Thus, a mechanism for holding on to the top of the stack while executing the subprogram and retrieving it afterwards is necessary.

Execution of ITER requires the same feature in a slightly different way. It consumes the list on top of the current stack. If the list is empty, it is dropped from the stack:

```
↓-ITER-NIL : ∀ {p-iter : Program (t :: txs) txs}
  ----------------------
  → [ Conf ce ([] :: xs) , ITER p-iter ]↓ xs
```

Otherwise the subprogram is applied to the first list element v and then the ITER instruction is reissued on the rest of the list vs and the current stack:

```
↓-ITER-CONS : ∀ {v : ⟦ t ⟧}{vs : ⟦ list t ⟧} {xs ys zs : Int txs} {p-iter : Program (t :: txs) txs}
  → [ Conf ce (v :: xs) , p-iter ]⇓ ys
  → [ Conf ce (vs :: ys) , ITER p-iter ]↓ zs
  ------------------------
  → [ Conf ce ((v :: vs) :: xs) , ITER p-iter ]↓ zs
```

The typing for ITER requires that the type of the underlying stack is preserved, but the subprogram *p-iter* is entitled to access and modify the stack beyond the first element x. Let's now consider stepwise execution. If the list on top has the form v :: vs, we need to stash the tail list vs somewhere while the subprogram processes the stack with v on top. After execution of the subprogram, we have to recover vs and try again with ITER.

As subprograms can be arbitrarily complex, in particular, they may contain DIP and ITER, we need a nestable solution. To this end, we add a single new instruction MPUSH1 that pushes a single value on the stack. This instruction is different from the normal PUSH instruction, which is limited to Pushable values that have a textual representation.

```
data ShadowInst {M : Type → Set} : Stack → Stack → Set where
  MPUSH1 : ∀{t : Type} → M t → ShadowInst rS (t :: rS)
```

We call the new instruction a *shadow instruction* because it does not appear in input programs. It is indexed by two stack types like any other instruction. A *shadow program* is defined like Program, but its first instruction can be a normal instruction or a shadow instruction. Shadow programs only appear at the top-level, never as subprograms nested in instructions. We elide the definition of ShadowProg as it is analogous to Program. Moreover, we provide a utility function mpush to generate a sequence of MUSH1 instructions from a list of values.

```
mpush : ∀ {M : Type → Set} {ri}{ro} {front : Stack}
  → All M front → ShadowProg{M} (front ++ ri) ro → ShadowProg{M} ri ro
mpush []       sp = sp
mpush (x :: xs) sp = mpush xs (MPUSH1 x • sp)
```

The small-step version of DIP n dp takes the top n elements from the stack and starts executing the program dp followed by the new instruction mpush *front* where *front* is the list of the n values that were removed from the stack.

```
prog-step ρ | DIP n dp ; p
  = record ρ { prg = dp ;• mpush (H.take n (stk ρ)) p ; stk = H.drop n (stk ρ) }
```

```
example-ITER : Program [ list nat ; nat ] [ nat ]
example-ITER = ITER (ADDnn ; end) ; end
```

Figure 4 Simple program using ITER.

Table 1 Program states during execution of Figure 4.

rSI	prg
[18 , 24] :: 0 :: []	ITER (ADD)
18 :: 0 :: []	ADD ; MPUSH [24]; ITER (ADD)
18 :: []	MPUSH [24]; ITER (ADD)
[24] :: 18 :: []	ITER (ADD)
24 :: 18 :: []	ADD ; MPUSH []; ITER (ADD)
42 :: []	MPUSH []; ITER (ADD)
[] :: 42 :: []	ITER (ADD)
42 :: []	end

The small-step version of ITER ip just pops the stack if the list is empty. Otherwise, if the top contains v :: vs, it pops this value, puts v on top of the stack and executes ip followed by mpush [vs] and then ITER ip and the rest of the program.

```
prog-step ρ | ITER ip ; p with stk ρ
... | [] :: rsi       = record ρ { prg = p ; stk = rsi }
... | (v :: vs) :: rsi = record ρ { prg = ip ;• (MPUSH1 vs • (ITER ip ; p)) ; stk = v :: rsi }
```

For illustration, Table 1 gives the stacks and shadow program of each intermediate state resulting from applying prog-step to the program in Figure 4 until program termination for the given input stack interpretation (omitting end for readability). This program adds a list of numbers on top of the stack to the number below.

3.3 Relation to Big-Step Semantics

Executing a program requires iterating the prog-step function. Our implementation drives this iteration by a step counter that is counted down at each instruction.

```
prog-step* : ℕ → CProgState ro → CProgState ro
prog-step* zero ρ = ρ
prog-step* (suc n) ρ = prog-step* n (prog-step ρ)
```

We prove that the original big-step semantics and our small-step semantics are equivalent in the usual sense.

```
bigstep⇒smallstep : ∀ (prg : ShadowProg txs tys)
  → [ Conf ce xs , prg ]⇓ ys
  → ∃[ n ] prog-step* n (cstate ce prg xs) ≡ cstate ce end ys

smallstep⇒bigstep : ∀ n → (prg : ShadowProg txs tys) → {xs : Int txs} {ys : Int tys}
  → prog-step* n (cstate ce prg xs) ≡ cstate ce end ys
  → [ Conf ce xs , prg ]⇓ ys
```

```
record PrgRunning (Mode : MODE) : Set where
  constructor pr
  field {pp ss x y} : Type
        self        : Contract Mode pp ss
        sender      : Contract Mode x y
        ρ           : ProgState Mode [ pair (list operation) ss ]

record Transaction (Mode : MODE) : Set where
  constructor _,_
  field pops    : (𝓜 Mode) (list operation)
        psender : Addr

data RunMode (Mode : MODE) : Set where
  Run  : PrgRunning Mode → RunMode Mode
  Cont : 𝓕 Mode → RunMode Mode
  Fail : 𝓖 Mode → RunMode Mode

record ExecState (Mode : MODE) : Set where
  constructor exc
  field accounts : Blockchain Mode
        MPstate  : RunMode Mode
        pending  : List (Transaction Mode)
```

Figure 5 Contract execution state.

3.4 Contract Execution and Execution Chains

The prog-step function can execute any Michelson program, not only those that comply to the typing restrictions of a contract. But it does not provide a mechanism to update the blockchain after successful contract execution nor one to execute other blockchain operations which might be emitted by a contract.

To implement these aspects of contract execution, the ProgState is augmented with further information as shown in Figure 5. The record PrgRunning holds the contracts involved in the current execution: self is the current contract and sender is the sender (the account that started the current contract). The ExecState holds the Blockchain, where contract execution results are saved, and a list of pending blockchain transactions to be executed. A value of type Transaction comprises a list of operations and the address of the sender of these operations. The field MPstate encodes the current mode of execution. Run indicates that a contract is currently executing the program in PrgRunning where we can take a step. Cont indicates the transition between one contract and the next; execution proceeds with the next pending blockchain operation. The $\mathcal{F}$ argument is used by the abstract execution to propagate information between contract invocations. Fail indicates a failure along with an error code in its $\mathcal{G}$ argument.

```
exec-step σ@(exc accts (Run (pr self _ (state en end [ new-ops , new-storage ] _))) pend)
  = record σ{ accounts = set (self-address en) (upd-storage self new-storage) accts
            ; MPstate = Cont tt
            ; pending  = (new-ops , self-address en) :: pend }
exec-step σ@(exc _ (Run ρr@(pr _ _ ρ)) _)
  = record σ{ MPstate = Run (record ρr{ ρ = prog-step ρ }) }
```

Figure 6 Program execution.

The function exec-step : CExecState → CExecState maps an execution state to its successor state just like prog-step did for program states. It only implements the features mentioned above that cannot be modeled by the program state alone. Its definition is too big to include it in full; instead we briefly explain its implementation, giving each case in the same order as in the implementation.

Figure 6 contains the cases when a contract is executing.

1. When execution of the current contract has terminated (i.e., MPstate is Run *pr* and ProgState.prg matches end), then intrinsic typing ensures that the stack interpretation contains the emitted blockchain operations *new-ops* paired with the new storage value *new-storage*. We add the emitted operations to the pending field, update the terminated contract's storage on the blockchain, and switch to RunMode Cont.
2. In all other cases of a running program, its ProgState evolves using prog-step.

In the remaining cases MPstate is Cont tt which means that no contract is currently executed. In this case pending is checked for other operations to be executed. Our model only implements the TRANSFER-TOKENS operation that initiates a new contract execution. We perform the following checks in this case:

- we fail unless the operation was emitted from a valid account;
- we fail unless the type of the parameter matches the input type of the called contract;
- we fail unless the target is a valid account;
- we fail unless the sender's balance contains sufficient tokens to support the transfer.

The first three cases can never occur during an actual execution of a Michelson smart contract execution chain: The TRANSFER-TOKENS instruction only works for values of type contract t, which ensures validity of the target address and that the parameter type is t. Moreover, operations can only be emitted by valid accounts. The checks are needed in our model because it does not maintain information about which addresses are valid contract addresses. We chose not to include this information as it adds complexity without contributing to our goal of proving the soundness of symbolic execution.

4 Dynamic Logic for Michelson

To obtain a dynamic logic suitable for symbolic execution we follow the Key approach [9] and extend first order logic with a modality $[p]$, where p is a program state. The intuitive meaning is that $[p]\Psi$ holds for a formula Ψ, if running p terminates in a state such that Ψ holds. That is, the formula $\Phi \to [p]\Psi$ has a similar meaning as the Hoare triple $\{\Phi\}\ p\ \{\Psi\}$.

In the following, we concentrate on the proof rules for the modality. For instance (and ignoring the details of the program state for now), $\Phi \to [end]\Psi \equiv \Phi \to \Psi$ if the program is empty. Many simple proof rules have the form $\Phi \to [i;p]\Psi \equiv \Phi_i \wedge \Phi \to [p]\Psi$ where the formula Φ_i describes the effect of instruction i. If the instruction is a branch instruction on a predicate Q, like `if` $Q\ p_1\ p_2$, the resulting formula is a disjunction as in $\Phi \to [(\texttt{if}\ Q\ p_1\ p_2);p]\Psi \equiv Q \wedge \Phi \to [p_1;p]\Psi \vee \neg Q \wedge \Phi \to [p_2;p]\Psi$.

```
data _⊢_ (Γ : Context) : Type → Set where
  var   : t ∈ Γ          → Γ ⊢ t
  const : ⟦ base bt ⟧     → Γ ⊢ base bt
  contr : ∀ {P : Passable t} → Addr → Γ ⊢ contract P
  func  : 1-func args result → Match Γ args → Γ ⊢ result

data Formula (Γ : Context) : Set where
  'false   : Formula Γ
  _:=_     : t ∈ Γ → Γ ⊢ t → Formula Γ
  _<ₘ_     : mutez ∈ Γ → mutez ∈ Γ → Formula Γ
  _≥ₘ_     : mutez ∈ Γ → mutez ∈ Γ → Formula Γ
```

Figure 7 Terms and formulas.

We start by defining the formulas of the logic in Subsection 4.1.

4.1 Terms and Formulas

At the core of any symbolic execution there are symbolic (i.e., logical) variables representing the unknown operands. We represent such variables by a typed deBruijn index into a given Context = List Type. An abstract stack is then a list of typed variables:

```
Match : Context → Stack → Set
Match Γ = All (_∈ Γ)
```

Any knowledge that we have about the values on the stack is encoded in the list of formulas (over the variables on the stack) that we maintain in the program state. Figure 7 shows the terms and formulas used for the logic. Term comprise variables, constants of base type and of contract type, and simple functions. Here, "function" stands for proper functions as well as data constructors. For convenience, we restrict function arguments to variables and rely on variable equality in the formulas to specify complex terms.

As an example for the interplay between context, stack, and formulas, suppose the context defines three variables of type nat like this Γ_1 = nat :: nat :: nat :: []. An abstract stack for this context might just contain a single variable x = 0∈, where the 0 ∈ refers to the first variable in Γ_1.

```
a-stack : Match Γ₁ (nat :: [])
a-stack = [ x ]
```

If we further want to enforce that $x = y + 3$ (on natural numbers), then we have to encode that in two simple formulas, one that associates 3 to variable v, and another that states $x = y + v$. We do not impose a constraint on y, so it serves as an unconstrained symbolic variable.

```
x=y+3 : List (Formula Γ₁)
x=y+3 = x := func 'ADDnn (y :: v :: [])
        :: v := const 3
        :: []
```

Formulas are mainly used to express equality of a variable with a term. The inequalities express the ordering on tokens. The latter is used for token transfers where we have to know that the sender has sufficiently many tokens to satisfy the requirements of the transfer. The reader may wonder about conjunction and disjunction: the proof rules only generate them in the form of a disjunction of conjunctions of simple formulas. We represent this structure as a list of lists of simple formulas. Repetition does not matter in this representation for two reasons: 1. disjunction and conjunction are both idempotent; 2. we are only interested in validity of a formula, but do not transform it in any way.

4.2 Representing Michelson Program State in DL

We simplify the handling of formulas of the form $\Phi \to [p]\Psi$ by reusing our previous definition of the type ProgState in a different mode as an *abstract* state.

```
AMode : Context → MODE
AMode Γ = record { ℳ = _∈ Γ
                 ; ℱ = List (Formula Γ)
                 ; 𝒢 = List (Formula Γ) ⊎ List (Formula Γ)
                 }
```

That is, we replace the normal representation of values in $\mathcal{M}$ by symbolic variables, in $\mathcal{F}$ we maintain a list (i.e., conjunction) of formulas, and in $\mathcal{G}$ we maintain a tagged list of formulas to represent different modes of failure.

The meaning of an abstract state is a conjunction that specifies the value for AMOUNT and BALANCE in the environment, it specifies the size of the stack and all values on it, and it collects further constraints generated by application of the proof rules.

Informally, an abstract program state represents $\Theta \Longrightarrow [prg]\Psi$ where

$$\Theta \equiv \textit{state of environment} = \mathsf{en} \wedge \textit{state of stack} = \mathsf{stk} \wedge \bigwedge_{\phi \in \Phi} \phi$$

The encoding of the implication in the abstract program state corresponds exactly to the abstract instance of the ProgState type (see Figure 3). Reusing the type in this way makes the formalization of symbolic execution very similar to the concrete execution model presented in Section 3. This similarity in turn makes the soundness proof easier and more concise. All constructs for concrete execution are reused in the abstract by instantiating their MODE parameter. Thus, they are automatically parameterized by a `Context` Γ and the names of the structures are the same as for concrete execution but prefixed with an α (only the abstract blockchain is called β`lockchain`).

Symbolic execution of control flow can lead to disjunctions over such states, which is represented using a list of abstract program states. Each of the branch comes with its own state, which requires existential quantification over the types of the variables in Γ.

```
⊎Prog-state : Stack → Set
⊎Prog-state ro = List (∃[ Γ ] αProg-state Γ ro)
```

Using Agda lists to represent conjunctions and disjunctions is convenient for two reasons.

1. Conjunctions and disjunctions do not mix: Φ always represents a conjunction over its elements and disjunctions can only occur as a result of some symbolic execution rules that implement control flow. In this case, the disjunction always affects every aspect of the abstract program state (i.e., the remaining programs will always differ).
2. Agda's "element of" relation for lists makes the implementation of the rules of the calculus simple and efficient.

4.3 Proof Rules for Michelson

The rules for symbolic execution are formalized by a function that maps an abstract program state into a set (list) of abstract program states.

```
αprog-step : ∀ {Γ ro} → αProg-state Γ ro → ⊎Prog-state ro
```

It mimics prog-step and gives a deterministic way of symbolic execution. Every (non-environmental) functional instruction can be executed concretely with a single rule as shown in Figure 6. During symbolic execution, the only thing that is guaranteed is that the stacks contain values of the expected type. For example, if the next instruction is ADDnn, we can conclude that there are two values of type nat on top of the stack before the instruction and one value of type nat afterwards. Moreover, we can say that this value is the sum of the two values that were on top of the stack before, but we have to express that with a constraint, i.e., a logical formula.

That is, symbolic execution of ADDnn introduces a new variable v_r that replaces the variables v_x and v_y from the top of the stack, and adds a clause that equates this new variable with the sum of the former two:

$$v_r := \text{ADDnn}\ v_x\ v_y$$

In this way, we can give a single symbolic execution rule for all functional instructions that return a single result.

```
αprog-step {Γ} (state αen (fct (D1 {result = result} f) ; prg) αst Φ)
  = [ (result :: Γ)
    , state (wkαE αen) (wkSP prg) (0∈ :: wkM (H.rest αst))
            (0∈ := wk⊢ (func f (H.front αst)) :: wkΦ Φ ) ]
```

Let's decompress this definition. We pattern match against the current (abstract) state to obtain the environment αen, the current instruction, the rest of the program prg, the stack αst, and the formula Φ. The constructor fct indicates a functional instruction and the constructor D1 indicates that f returns a single result of type $result$.

As the instruction does not implement any control flow, there is only a single next state. Its description starts with the extended context $result :: \Gamma$, which introduces a new variable of type $result$ for the result. The name, rather the deBruijn address, of this variable is 0∈, which denotes the first entry in the context. The second component describes the new state, which (ignoring the wk functions for the moment) keeps the environment, moves to the rest of the program, pushes the result on the stack after removing the arguments using H.rest, and pushes a new equation that defines the value of 0∈ as the result of applying f to the front of the stack. The functions H.front and H.rest operate on heterogenous lists and are defined such that $\alpha st \equiv$ H.front αst H.++ H.rest αst where the actual division is driven by the type of f. The operation H.++ is concatenation of heterogenous lists. The wk functions are a consequence of using deBruijn indices for variables: if we introduce new variables, all existing variables have to be incremented by the number of new variables (i.e., weakened). We do not show their definition, as this manipulation of deBruijn indices is standard.

We do not have a general mechanism for the other functional instructions (see Figure 8), as they behave very differently in a symbolic context: UNPAIR requires two new variables and clauses, while SWAP only changes the position of two stack values. No new variables or clauses are necessary because SWAP only reconfigures the stack.

The instruction PUSH needs special treatment because it can handle arbitrarily complex compound values. When pushing a value x of primitive type, it is sufficient to add a new variable and a clause which sets this variable equal to the term const x. But if x has a list

```
αprog-step {Γ} (state αen (fct (Dm ('UNPAIR {t1} {t2})) ; prg) (p∈ :: αst) Φ)
  = [ (t1 :: t2 :: Γ)
    , state (wkαE αen) (wkSP prg) (0∈ :: 1∈ :: wkM αst)
            ( 0∈ := func 'CAR [ wk∈ p∈ ] :: 1∈ := func 'CDR [ wk∈ p∈ ] :: wkΦ Φ ) ]

αprog-step α@(state αen (fct (Dm 'SWAP) ; prg) (x∈ :: y∈ :: αst) Φ)
  = [ -, record α{ prg = prg ; stk = y∈ :: x∈ :: αst } ]

αprog-step {Γ} (state αen (fct ('PUSH P x) ; prg) αst Φ)
  = [ (expandΓ P x ++ Γ)
    , state (wkαE αen) (wkSP prg) ((∈wk (0∈exΓ P)) :: wkM αst)
            (Φwk (unfold P x) ++ wkΦ Φ) ]
```

Figure 8 Functional instructions (excerpt).

type or an option type, its value cannot be expressed with a const term. In general, the symbolic execution of a single PUSH instruction may create arbitrarily many (but linear in the size of the pushed value) new variables and clauses.

To this end, the function unfold P x creates all clauses required to express the value x. This process defines a list of new variables of types defined by expandΓ P x.[6] For example, PUSH {list ty} P (y :: ys) gives rise to two new variables r_y of type ty for y and r_{ys} of type list ty for ys and an equation r := func (CONS $[r_y, r_{ys}]$), where r is the variable for the result. The function unfold proceeds recursively: if ys = [], its variable can be set to func (NIL ty) [], otherwise it will be further decomposed. Similarly for y: if ty is a primitive type, it can be set to const y, otherwise it must be further decomposed as well.

As an example, we show the result of unfolding the list $[0, 1]$: list nat. The generated context is Γ_2 = list nat :: list nat :: list nat :: nat :: nat :: [] and the generated list of equations to represent the list is as follows.

```
eqn : List (Formula Γ₂)
eqn = c₁ := func 'CONS (x₀ :: c₂ :: [])
    :: c₂ := func 'CONS (x₁ :: c₃ :: [])
    :: c₃ := func ('NIL nat) []
    :: x₀ := const 0
    :: x₁ := const 1
    :: []
```

We finish with the abstract execution of the conditional instruction IF-NONE (see Figure 9). This instruction expects a value of type option t on top of the stack. Here we have two possible next states, depending on whether the value is present. The first disjunct deals with the case where the value is NONE. In this case, the stack is popped, the *thn* branch is taken, and the equation enforcing the value to be NONE is added. There are no new variables, so there is no weakening in this disjunct.

[6] We do not include the tedious definitions of these auxiliary functions here, but encourage the interested reader to check the supplementary material.

```
αprog-step {Γ} (state αen (IF-NONE {t = t} thn els ; prg) (oe :: αst) Φ)
  = [ Γ        , state αen           (thn ;• prg)       αst (oe := func ('NONE t) [] :: Φ)
    ; (t :: Γ) , state (wkαE αen) (els ;• wkSP prg) (0e :: wkM αst)
                       (wke oe := func 'SOME [ 0e ] :: wkΦ Φ) ]
```

Figure 9 Symbolic execution of IF-NONE.

The second disjunct models the case where the value on top of the stack is SOME y. Here we need a new variable of type t for y, pop the stack and push the new variable, we take the *els* branch, and add an equation that forces the value to be SOME y.

4.4 Proof Rules for the Blockchain Run-time

Just like the symbolic execution rules for the Michelson DL, those for the DL on blockchain operations are given analogously.

```
⊎ExecState : Set
⊎ExecState = List (∃[ Γ ] αExecState Γ)
```

```
αexec-step : ∀ {Γ} → αExecState Γ → ⊎ExecState
```

The switch from concrete to abstract execution state is achieve by changing the *Mode* parameter of the ExecState (see Figure 5). Its $\mathcal{F}$ field replaces concrete semantics by abstract semantics throughout all state components.

Unfortunately αexec-step cannot represent exec-step exactly, if MPstate is Cont Φ, that is: a contract has terminated and we need to check the pending field for further operations to be executed. At this point, the predicate Φ has to supply sufficient information about the values of the variables representing the pending operations to proceed in a meaningful way. The pending list contains pairs of a list of operations and a sender address. While the latter is a concrete address, the former is a variable of type `list operation ∈ Γ`. To proceed, we have to know if the list is empty (so that we can proceed to the next block of pending operations) or not. In the latter case, we need to ensure that the first element of the operation list is a TRANSFER-TOKENS, and so on.

To this end, we defined several auxiliary functions to inspect the constraints in Φ for patterns that restrict the models sufficiently. For example, the function find-tt-list takes a conjunction of formulas and a variable of type list t and tries to find a formula that restricts this variable to NIL or CONS:

```
find-tt-list : ∀ {Γ}{t} → List (Formula Γ) → list t ∈ Γ
  → Maybe (Match Γ [] ⊎ Match Γ [ t ; list t ])
```

```
find-tt-list-soundness : ∀ {Γ}{t} → (Φ : List (Formula Γ)) → (le : list t ∈ Γ)
  → find-tt-list Φ le ≡ just (inj₁ [])
  → ∀ (γ : Int Γ) → γ ⊨Φ Φ
  → lookup γ le ≡ []
```

```
val⊢ : ∀ {ty Γ} → Int Γ → Γ ⊢ ty → ⟦ ty ⟧
val⊢ γ (var v∈)       = lookup γ v∈
val⊢ γ (const b)      = b
val⊢ γ (contr adr)    = adr
val⊢ γ (func f args) = appD1 f (map (lookup γ) args)

_⊨φ_ : ∀ {Γ} → Int Γ → Formula Γ → Set
γ ⊨φ 'false          = ⊥
γ ⊨φ (v∈ := trm) = lookup γ v∈ ≡ val⊢ γ trm
γ ⊨φ (x <ₘ x₁)    = lookup γ x < lookup γ x₁
γ ⊨φ (x ≥ₘ x₁)    = lookup γ x ≥ lookup γ x₁
```

Figure 10 Semantics of terms and formulas.

We only show the soundness lemma for NIL, as the one for CONS is analogous. This approach is not complete as the implementation of find-tt-list is tailored to the constraints as they are produced by symbolic execution.

The full implementation is quite involved and relies on several further lemmas that examine constraints (for example if the current balance of a sender is sufficient for a token transfer) in a similar way. We refer the interested reader to the supplement.

The remaining cases deal with a terminated contract execution where the new state is written back to the blockchain or the execution of an abstract program step for the contract under execution. The first case is similar to the concrete implementation where new variables are introduced for the updated values. The second case is more complicated because the context extensions from the abstract program step are encoded in the list of resulting disjunctions, so an additional term has to be supplied proving that these contexts are actually an extension of the original context.

5 Semantics and Soundness

5.1 Values and Models

As a context is just a list of types like a stack, its interpretation is also a heterogeneous list of values as defined by Int. For a given context interpretation γ, the semantics of a term and a formula is defined as usual (see Figure 10).

For a given context interpretation γ and abstract and concrete (program or execution) states, the predicates modρ and modσ express that under this interpretation the given abstract state models the concrete state. This is the case when the formulas in Φ are true under γ and the real and variable values are the same for the stacks and every other element.

```
MODELING : Context → (MODE → Set) → Set₁
MODELING Γ F = Abstract F Γ → Concrete F → Set

modρ : ∀ {Γ} → Int Γ → MODELING Γ λ M → Prog-state M ro
modρ γ (state {ri = αri} αen αprg rVM Φ)
       (state {ri} en prg stk tt)
  = Σ (αri ≡ ri) λ{ refl →
    modE γ αen en × modprg γ αprg prg × modS γ rVM stk × modΦ γ Φ}
```

```
soundness γ (state αen (IF-NONE thn els ; aprg) (oe :: rVM) Φ)
            (state en (.IF-NONE thn els ; cprg) (just x :: stk) tt)
            (modρ⟨ mE , (o≡ , mrS) , (refl , refl , mPRG) , mΦ ⟩)
  = _ , [ x ] , _ , 1∈ , (refl , wkmodE mE , modprg-extend els (wkmodprg mPRG) ,
    (refl , wkmodS mrS) , (o≡ , wkmodΦ mΦ))
soundness γ (state αen (IF-NONE thn els ; aprg) (oe :: rVM) Φ)
            (state en (.IF-NONE thn els ; cprg) (nothing :: stk) tt)
            (modρ⟨ mE , (o≡ , mrS) , (refl , refl , mPRG) , mΦ ⟩)
  = _ , [] , _ , 0∈ , (refl , mE , modprg-extend thn mPRG , mrS , (o≡ , mΦ))
```

Figure 11 Prog-step soundness for IF-NONE (excerpt).

They all have a similar structure expressed by the MODELING function as they relate an abstract thing with a concrete thing. They are implemented by several auxiliary `mod`X predicates for every subcomponent of program and execution states. For example, ModE relates execution environments, modprg relates shadow programs, modS relates stacks, and modΦ checks that the formulas are all true. The definition of modσ is similar.

To show that a disjunction of abstract states models a concrete state, we show that one of the states in the disjunction models the state:

```
mod⊎σ : ∀ {Γ} → Int Γ → ⊎ExecState → CExecState → Set
mod⊎σ {Γ} γ ⊎σ σ = ∃[ ασ ] (Γ , ασ) ∈ ⊎σ × modσ γ ασ σ
```

5.2 Soundness of the DL

We prove the soundness of the logic by showing that when an abstract state models a concrete one, the result of one-step symbolic execution models the result from concrete execution of the same step. Here are the types of the proof terms for program steps and execution steps.

```
soundness : ∀ {Γ ro} γ αρ ρ → modρ {ro} {Γ} γ αρ ρ
          → ∃[ Γ‘ ] ∃[ γ‘ ] mod⊎ρ {Γ = Γ‘ ++ Γ} (γ‘ H.++ γ) (αprog-step αρ) (prog-step ρ)
```

```
soundness : ∀ {Γ} (γ : Int Γ) → ∀ ασ σ → modσ γ ασ σ
          → ∃[ Φ ] ExecState.MPstate ασ ≡ APanic Φ
          ⊎ ∃[ Γ‘ ] ∃[ γ‘ ] mod⊎σ {Γ‘ ++ Γ} (γ‘ H.++ γ) (αexec-step ασ) (exec-step σ)
```

The first soundness statement addresses soundness of α`prog-step`. As the modeling relation is mostly composed of equalities, the proof gets accepted by Agda, once we supply sufficiently precise arguments to match the cases in the definition of αprog-step. We pattern match against refl and parts of the arguments, as well as we show that the weakened parts of the formula are still modeled with the extended context (if new variables were introduced in the case).

Figure 11 shows the case for the IF-NONE instruction. Without going into details, it is easy to spot the handling of the concrete and abstract stack and that the outcome of the test determines which of the possibilities of the abstract outcome is chosen (cf. $0 \in$ and $1 \in$).

The most complicated case of this proof establishes soundness for any scalar function (see Figure 12). It works by showing that applying the front of the previous stack interpretation to the given function yields the same result as applying the extended interpretation of the top of the previous stack matching to it.

```
soundness γ (state αen (fct (D1 f) ; aprg) rVM Φ)
            (state en (.fct (D1 f) ; cprg) stk tt)
            (modρ⟨ mE , mrS , (refl , refl , mPRG) , mΦ ⟩)
  with modS++ rVM stk mrS
... | mfront , mrest =
  let result = appD1 f (H.front stk) in
  _ , [ result ] , _ , 0∈ , (refl ,
  wkmodE mE , wkmodprg mPRG , (refl , wkmodS mrest) ,
  (cong (appD1 f) (trans (sym (modIMI mfront)) (wkIMI {γ' = [ result ]}))) , wkmodΦ mΦ))
```

Figure 12 Prog-step soundness for scalar functions (excerpt).

The second soundness statement establishes soundness for those cases of αexec-step where a contract execution is active. This part appears simple because it only covers two cases: Either we are in the middle of running a contract, in which case we reuse the soundness proof for program state execution, or the current contract execution has terminated and we have to prove that the blockchain and the pending list are updated correctly. The first case is straightforward, but tedious because we need to copy parts of the previous proof. The second case is fairly technical as it involves getting the proof in sync with the definitions of concrete and abstract execution.

6 Related Work

Research on formal verification of blockchain-based applications has experienced rapid growth in the last decade. Various techniques and frameworks have been applied to enhance the safety of smart contracts. In this section, we discuss some key approaches, particularly those employing symbolic execution in the context of smart contracts.

6.1 Verification of Smart Contracts

Symbolic execution is a powerful technique for systematically exploring program paths and identifying potential vulnerabilities in smart contracts. Most of the existing tools focus on the Ethereum platform. Tsankov et al. introduced SECURIFY [32], a tool that utilizes symbolic execution to perform practical security analysis on Ethereum smart contracts. It targets common vulnerability security patterns specified in a designated domain-specific language. SECURIFY symbolically encodes the dependence graph of the contract in stratified Datalog to extract semantic information from the code. After obtaining semantic facts, it checks whether the security patterns hold or not. Similarly, Manticore [26] and KEVM [18] use symbolic execution to analyze Ethereum smart contracts. KEVM is an executable formal specification built with the K Framework for the Ethereum virtual machine's bytecode (EVM), a stack-based and low-level smart contract language for the Ethereum blockchain. Since tokens can hold a significant amount of value, they are often targeted for attacks. Therefore, several tools [18, 29] conduct case studies for the implementations of token standards.

Several approaches use existing formal verification frameworks to ensure the correctness and security of smart contracts. Amani et al. [3] proposed the formal verification of Ethereum smart contracts in Isabelle/HOL. Hirai [19] formalizes the EVM using Lem, a language to specify semantic definitions. The formal verification of smart contracts is achieved using

the Isabelle proof assistant. Mi-cho-Coq [10] is a framework for the proof assistant Coq to verify functional correctness of Michelson smart contracts. They formalize the semantics of a Michelson in Coq using a weakest precondition calculus and verify several contracts. It provides full coverage of the language whereas our goal is to give a blueprint for a soundness proof of symbolic execution.

There are several tools for automated verification including solc-verify [15], VerX [4], and Oyente [22]. solc-verify processes smart contracts written in Solidity and discharges verification conditions using modular program analysis and SMT solving. It operates at the level of the contract source code, with properties specified as contract invariants and function pre- and post-conditions provided as annotations in the code by the developer. This approach offers a scalable, automated, and user-friendly formal verification solution for Solidity smart contracts. The core of solc-verify involves translating Solidity contracts to Boogie IVL (Intermediate Verification Language), a language designed for verification.

Nishida et al. [27] developed HELMHOLTZ, an automated verification tool for Michelson. While both research efforts aim to build a verification tool for smart contracts written in Michelson, HELMHOLTZ is based on refinement types, whereas we consider symbolic execution. HELMHOLTZ has better coverage of Michelson instructions than we currently have, but it can only verify a single contract whereas our model and soundness proof covers full inter-contract verification. The HELMHOLTZ developers plan to extend Helmholtz with inter-contract behavior.

Bau et al. [8] implement a static analyzer for Michelson within the modular static analyser MOPSA that is based on abstract interpretation. It is able to infer invariants on a contract's storage over several calls and it can prove the absence of errors at run time.

Da Horta et al. [5] aim at automating as much of the verification process as possible by automatically translating a Michelson contract into an equivalent program for the deductive program verification platform WHY3. However, they found that sometimes user intervention was required, and their tool can only verify single contracts individually.

6.2 Symbolic Execution for Bytecode Interpretation

As there are some parallels between Michelson and bytecode languages, we discuss symbolic execution methods for some seleted bytecode languages.

Albert et al. [2] transform Java bytecode into a logic program to utilize analysis techniques from logic programming, specifically symbolic semantics, for the formal verification of the bytecode. They verify properties such as termination and run-time error freeness and infer resource bounds. The dynamic aspects of bytecode, such as control flow and data flow, are effectively handled through the analysis performed on the logic program. Balasubramanian, Daniel et al. [7] include dynamic symbolic execution tailored for Java-based web server environments. Their tool analyzes the bytecode interactions within the Java Virtual Machine and focuses on bytecode instructions, method calls, object manipulations and memory interactions to detect vulnerabilities and bugs.

Several approaches address formal semantics and analysis for WebAssembly (Wasm) [24, 21, 34]. Watt, Conrad et al. [34] present Wasm Logic, a formal program logic for WebAssembly. The authors mechanize Wasm logic and its proof of correctness in Isabelle/HOL. To this end, they propose an alternative semantics. Just like our work (we propose a logic on an alternative semantics, mechanize it, and prove its soundness in Agda), their aim is to provide a logical basis for static analysis tools.

Marques et al. [24] present a concolic execution engine that systematically explores different program paths by combining concrete and symbolic execution to enable automated testing and fault detection. It models execution behavior and uses constraint solvers to

generate inputs and explore paths, taking into account the complexity of Wasm's stack-based execution and binary format. Unlike our work, their work is geared towards implementing a realistic tool.

6.3 Related Uses of Dynamic Logic

The idea of using dynamic logic for symbolic execution can be traced back to Heisel et al. [17]. They formalize Burstall's verification method [11] using symbolic execution and induction in the framework of dynamic logic.

Maingaud et al. [23] define a program logic for imperative ML programs based on dynamic logic and prove its soundness. Their goal is to use this logic as a basis for symbolic execution.

Similar to our approach, the research of Ahrendt et al. [1] emphasizes data integrity in Solidity smart contracts. This framework verifies smart contracts and ensures strong data integrity and functional correctness under various conditions. It introduces a specification language for defining contract properties and behaviors that are critical for security and reliability. Similar approaches to ours aim to verify the correctness and security of smart contracts, but differ in methodology and target languages. Their approach uses dynamic logic for invariant-based specifications with prototype-based tools, while our approach uses dynamic logic for symbolic execution and focuses on formal proofs.

Abstract execution [31] is a static verification framework based on symbolic execution. It is geared at schematic programs, i.e., programs with placeholders for program fragments, so that it can be used to prove certain program transformations correct. Its logical basis is dynamic logic extending earlier work for Java [9].

7 Conclusion

We presented a dynamic logic for Michelson as well as its extension to blockchain operations on a small but representative subset of Michelson. The goal was to create a core model that covers instances of all kinds of operations and that can be easily extended with further Michelson instructions. We achieved full coverage of scalar functional instructions, the majority of Michelson instructions. To include any further scalar instruction, one only has to specify its typing rule and its implementation in Agda. The symbolic execution rule and the soundness proof for that rule is already provided by our model. Further instructions that retrieve information from the execution environment can be added easily as well by extending the Environment record and its subcomponents to include such information.

We cover three exemplary instructions for control flow, because most other conditional and looping instructions are either very similar or very simple and thus easy to include in the presented model. One aspect of Michelson that is not covered is first-class functions. Including them might require some reworking of the current model to store such values on the stack.

Efficient symbolic execution is **not** a goal of this work: Agda can normalize a concrete or symbolic execution state to enable inspection of the state after one or more execution steps, but in our experiments normalization was sometimes infeasible after less than ten symbolic execution steps. Nevertheless, we plan to use our soundness proof as the basis for an efficient symbolic interpreter for Michelson in ongoing work.

References

1 Wolfgang Ahrendt and Richard Bubel. Functional verification of smart contracts via strong data integrity. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 9–24, Cham, 2020. Springer International Publishing.

2 Elvira Albert, Miguel Gómez-Zamalloa, Laurent Hubert, and Germán Puebla. Verification of java bytecode using analysis and transformation of logic programs. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, pages 124–139, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

3 Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 66–77, January 2018. doi:10.1145/3167084.

4 Permenev Anton, Dimitrov Dimitar, Tsankov Petar, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677, 2020. doi:10.1109/SP40000.2020.00024.

5 Luís Pedro Arrojado da Horta, João Santos Reis, Simão Melo de Sousa, and Mário Pereira. A tool for proving michelson smart contracts in why3. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 409–414, 2020. doi:10.1109/Blockchain50366.2020.00059.

6 Barnabas Arvay, Thi Thu Ha Doan, and Peter Thiemann. Contract Orchestration for Michelson. Software, version 0.5 (visited on 2024-08-29). URL: https://freidok.uni-freiburg.de/data/255176.

7 Daniel Balasubramanian, Zhenkai Zhang, Dan McDermet, and Gabor Karsai. Dynamic symbolic execution for the analysis of web server applications in java. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 2178–2185, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3297280.3297494.

8 Guillaume Bau, Antoine Miné, Vincent Botbol, and Mehdi Bouaziz. Abstract interpretation of michelson smart-contracts. In *Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2022, pages 36–43, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3520313.3534660.

9 Bernhard Beckert, Vladimir Klebanov, and Benjamin Weiß. Dynamic logic for java. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors, *Deductive Software Verification – The KeY Book: From Theory to Practice*, pages 49–106. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-49812-6_3.

10 B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson. Mi-Cho-Coq, a framework for certifying Tezos smart contracts. In *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part I*, volume 12232 of *Lecture Notes in Computer Science*, pages 368–379. Springer, 2019. doi:10.1007/978-3-030-54994-7_28.

11 Rod M. Burstall. Program proving as hand simulation with a little induction. In Jack L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 308–312. North-Holland, 1974.

12 Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

13 Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 281–290. ACM, 2008. doi:10.1145/1368088.1368127.

14 L. Goodman. Tezos-a self-amending crypto-ledger, 2014. URL: `https://www.tezos.com/static/papers/white-paper.pdf`.

15 Á. Hajdu and D. Jovanović. solc-verify: A modular verifier for solidity smart contracts. In S. Chakraborty and J. A. Navas, editors, *Verified Software. Theories, Tools, and Experiments*, pages 161–179. Springer International Publishing, 2020.

16 David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.

17 Maritta Heisel, Wolfgang Reif, and Werner Stephan. Program verification by symbolic execution and induction. In Katharina Morik, editor, *GWAI-87, 11th German Workshop on Artificial Intelligence, Geseke, Germany, September 28 - October 2, 1987, Proceedings*, volume 152 of *Informatik-Fachberichte*, pages 201–210. Springer, 1987. `doi:10.1007/978-3-642-73005-4_22`.

18 Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A complete formal semantics of the Ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018. `doi:10.1109/CSF.2018.00022`.

19 Y. Hirai. Defining the Ethereum virtual machine for interactive theorem provers. In *Financial Cryptography and Data Security*, pages 520–535. Springer International Publishing, 2017.

20 James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. `doi:10.1145/360248.360252`.

21 Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 1045–1058, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3297858.3304068`.

22 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 254–269, 2016.

23 Séverine Maingaud, Vincent Balat, Richard Bubel, Reiner Hähnle, and Alexandre Miquel. Specifying imperative ML-like programs using dynamic logic. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *Lecture Notes in Computer Science*, pages 122–137. Springer, 2010. `doi:10.1007/978-3-642-18070-5_9`.

24 Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão. Concolic Execution for WebAssembly. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2022.11`.

25 Michelson: The language of smart contracts in Tezos. URL: `https://tezos.gitlab.io/alpha/michelson.html`.

26 Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189, 2019. `doi:10.1109/ASE.2019.00133`.

27 Yuki Nishida, Hiromasa Saito, Ran Chen, Akira Kawata, Jun Furuse, Kohei Suenaga, and Atsushi Igarashi. HELMHOLTZ: A verifier for Tezos smart contracts based on refinement types. *New Generation Computing*, 40:507–540, 2022. `doi:10.1007/s00354-022-00167-1`.

28 Nomadic Lab. Michelson: the language of smart contracts in tezos, 2018-2023. Last accessed 17 October 2023. URL: `https://tezos.gitlab.io/michelson-reference/`.

29 Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A formal verification tool for Ethereum VM bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on*

European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 912–915, October 2018. doi:10.1145/3236024.3264591.

30 Corina S. Pasareanu. *Symbolic Execution: The Basics*, pages 5–20. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-031-02551-8_2.

31 Dominic Steinhöfel and Reiner Hähnle. Abstract execution. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *Lecture Notes in Computer Science*, pages 319–336. Springer, 2019. doi:10.1007/978-3-030-30942-8_20.

32 Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, October 2018. doi:10.1145/3243734.3243780.

33 Philip Wadler, Wen Kokke, and Jeremy G. Siek. Programming language foundations in Agda, August 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.

34 Conrad Watt, Petar Maksimović, Neelakantan R. Krishnaswami, and Philippa Gardner. A Program Logic for First-Order Encapsulated WebAssembly. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:30, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2019.9.