New Generation Computing (2022) 40:507–540
https://doi.org/10.1007/s00354-022-00167-1

NEW GENERATION COMPUTING

HELMHOLTZ: A Verifier for Tezos Smart Contracts Based on Refinement Types

Yuki Nishida[1] · Hiromasa Saito[1] · Ran Chen[1] · Akira Kawata[1] · Jun Furuse[2] · Kohei Suenaga[1] · Atsushi Igarashi[1]

Received: 1 August 2021 / Accepted: 23 March 2022 / Published online: 27 May 2022

Abstract

A *smart contract* is a program executed on a blockchain, based on which many cryptocurrencies are implemented, and is being used for automating transactions. Due to the large amount of money that smart contracts deal with, there is a surging demand for a method that can statically and formally verify them. This article describes our type-based static verification tool HELMHOLTZ for Michelson, which is a statically typed stack-based language for writing smart contracts that are executed on the blockchain platform Tezos. HELMHOLTZ is designed on top of our extension of Michelson's type system with refinement types. HELMHOLTZ takes a Michelson program annotated with a user-defined specification written in the form of a refinement type as input; it then typechecks the program against the specification based on the refinement type system, discharging the generated verification conditions with the SMT solver Z3. We briefly introduce our refinement type system for the core calculus Mini-Michelson of Michelson, which incorporates the characteristic features such as compound datatypes (e.g., lists and pairs), higher-order functions, and invocation of another contract. HELMHOLTZ successfully verifies several practical Michelson programs, including one that transfers money to an account and that checks a digital signature.

Keywords Smart contract · Blockchain · Formal verification · Tools

Introduction

A *blockchain* is a data structure to implement a distributed ledger in a trustless yet secure way. The idea of blockchains is initially devised for the Bitcoin cryptocurrency [13] platform. Many cryptocurrencies are implemented using blockchains, in which value equivalent to a significant amount of money is exchanged.

✉ Yuki Nishida
nishida@fos.kuis.kyoto-u.ac.jp

Extended author information available on the last page of the article

Recently, many cryptocurrency platforms allow programs to be executed on a blockchain. Such programs are called *smart contracts* [20] (or, simply *contracts* in this article) since they work as a device to enable automated execution of a contract. Technically speaking, a smart contract is a program P_a associated with an account a on a blockchain. (The word *contract* is also used to denote the account with which a smart contract is associated.) When the account a receives money from another account b with a parameter v, the computation defined in P_a is conducted, during which the state of the account a (e.g., the balance of the account and values that are stored by the previous invocations of P_a) which is recorded on the blockchain may be updated. The contract P_a may execute money transactions to another account (say c), which result in invocations of other contracts (say P_c) during or after the computation; therefore, contract invocations may be chained.

Although smart contracts' original motivation was handling simple transactions (e.g., money transfer) among the accounts on a blockchain, recent contracts are being used for more complicated purposes (e.g., establishing a fund involving multiple accounts). Following this trend, the languages for writing smart contracts also evolve from those that allow a contract to execute relatively simple transactions (e.g., Script for Bitcoin) to those that allow a program that is as complex as one written in standard programming languages (e.g., EVM for Ethereum and Michelson [14] for Tezos [6]).

Due to a large amount of money they deal with, verification of smart contracts is imperative. *Static* verification is especially needed since a smart contract cannot be fixed once deployed on a blockchain. Attack on a vulnerable contract indeed happened. For example, the DAO attack, in which the vulnerability of a fundraising contract was exploited, resulted in the loss of cryptocurrency equivalent to approximately 150M USD [19].

In this article, we describe our type-based static verifier Helmholtz[1] for smart contracts written in Michelson. The Michelson language is a statically and simply typed stack-based language equipped with rich data types (e.g., lists, maps, and higher-order functions) and primitives to manipulate them. Although several high-level languages that compile to Michelson are being developed, Michelson is most widely used to write a smart contract for Tezos as of writing.

A Michelson program expresses the above computation in a purely functional style, in which the Michelson program corresponding to P_a is defined as a function. The function takes a pair of the parameter v and a value s that represents the current state of the account (called *storage*) and returns a pair of a list of *operations* and the updated storage s'. Here, an operation is a Michelson value that expresses the computation (e.g., transferring money to an account and invoking the contract associated with the account) that is to be conducted after the current computation (i.e., P_a) terminates. After the computation specified by P_a finishes with a pair of an operation list and a storage value, a blockchain system invokes the computation specified in the operation list. This purely functional

[1] Hermann von Helmholtz (1821–1894), a German physicist and physician, was a doctoral advisor of Albert A. Michelson (1852–1931), whom the Michelson language is apparently named after.

style admits static verification methods for Michelson programs similar to those for standard functional languages.

As the theoretical foundation of Helmholtz , we design a refinement type system for Michelson as an extension of the original simple type system. In contrast to standard refinement types that refine the types of values, our type system refines the type of *stacks*.

We show that our tool can verify several practical smart contracts. In addition to the contracts we wrote ourselves, we apply our tool to the sample Michelson programs used in Mi-cho-coq [3], a formalization of Michelson in Coq proof assistant [22]. These contracts consist of practical contracts such as one that checks a digital signature and one that transfers money.

We note that Helmholtz currently supports approximately 80% of the whole instructions of the Michelson language. Another limitation of the current Helmholtz is that it can verify only a single contract, although one often uses multiple contracts for an application, in which a contract may call another by a money transfer operation, and their behavior as a whole is of interest. We are currently extending Helmholtz so that it can deal with more programs.

Our contribution is summarized as follows: (1) Definition of the core calculus Mini-Michelson and its refinement type system; (2) Automated verification tool Helmholtz for Michelson contracts implemented based on the type system of Mini-Michelson ; the interface to the implementation can be found at https://www.fos.kuis.kyoto-u.ac.jp/trylang/Helmholtz; and (3) Evaluation of Helmholtz with various Michelson contracts, including practical ones. A preliminary version of this article was presented at International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) in 2021. We have given detailed proofs of properties of Mini-Michelson and a more detailed description about the verifier implementation, in addition to revision of the text.

The rest of this article is organized as follows. Before introducing the technical details, we present an overview of the verifier Helmholtz in the next section using a simple example of a Michelson contract. The following section introduces the core calculus Mini-Michelson with its refinement type system and states soundness of the refinement type system. (Detailed proofs are deferred to Appendix A.) We also discuss a few extensions implemented in the verifier. The next section describes the verifier Helmholtz , a case study, and experimental results. After discussing related work in the next section, we conclude in the last section.

Overview of Helmholtz and Michelson

We give an overview of our tool Helmholtz in this section before presenting its technical details. We also explain Michelson by example (An Example Contract in Michelson) and user-written annotation added to a Michelson program for verification purposes (Specification).

```
parameter unit;
storage unit;
<< ContractAnnot { (param, st) | True } ->
   { (ops, st') | amount = 0 && ops = [] ||
      amount <> 0 && (match contract_opt source with
                      | Some c -> ops = [Transfer Unit amount c]
                      | None -> False) }
   & { _ | False } >>
code                        /* (param,st) */
 { CDR;                     /* st */
   NIL operation;           /* [] ▷ st */
   AMOUNT;                  /* amount ▷ [] ▷ st */
   PUSH mutez 0;            /* 0 ▷ amount ▷ [] ▷ st */
   IFCMPEQ
    {}                      /* [] ▷ st   (amount ≤ 0) */
    {                       /* [] ▷ st   (amount > 0) */
      SOURCE;               /* src ▷ [] ▷ st */
      CONTRACT unit;        /* Some (Contract src) ▷ [] ▷ st */
      ASSERT_SOME;          /* (Contract src) ▷ [] ▷ st */
      AMOUNT; UNIT;
      /* Unit ▷ amount ▷ (Contract src) ▷ [] ▷ st */
      TRANSFER_TOKENS;
      /* (Transfer Unit amount (Contract src)) ▷ [] ▷ st */
      CONS
      /* [Transfer Unit amount (Contract src)] ▷ st */
    };
    /* ops ▷ st, where ops is the top element at the end of each branch, namely,
       [Transfer Unit amount (Contract src)] if amount > 0; or [] otherwise */
   PAIR }                   /* (ops, st) */
```

Fig. 1 `boomerang.tz`. The comment inside /* */ describes the stack at the program point

Helmholtz

As input, Helmholtz takes a Michelson program annotated with (1) its specification expressed in a refinement type and (2) additional user annotations such as loop invariants. It typechecks the annotated program against the specification using our refinement type system; the verification conditions generated during the typechecking is discharged by the SMT solver Z3 [4]. If the code successfully typechecks, then the program is guaranteed to satisfy the specification.

Helmholtz is implemented as a subcommand of `tezos-client`, the client program of the Tezos blockchain. For example, to verify `boomerang.tz` in Fig. 1, we run `tezos-client refinement boomerang.tz`. If the verification succeeds, the command outputs `VERIFIED` to the terminal screen (with a few log messages); otherwise, it outputs `UNVERIFIED`.

An Example Contract in Michelson

Figure 1 shows an example of a Michelson program called `boomerang`. A Michelson program is associated with an account on the Tezos blockchain; the program

is invoked by transferring money to this account. This artificial program in Fig. 1, when it is invoked, is supposed to transfer the received money back to the account that initiated the transaction.

A Michelson program starts with type declarations of its `parameter`, whose value is given by contract invocation, and `storage`, which is the state that the contract account stores. Lines 1–2 declare that the types of both are `unit`, the type inhabited by the only value `Unit`. Lines 3–8 surrounded by << and >> are a user-written annotation used by Helmholtz for verification; we will explain this annotation later. The `code` section in Lines 10–29 is the body of this program.

Let us take a look at the `code` section of the program. In the following explanation of each instruction, we describe the state of the stack after each instruction as comments; stack elements are delimited by ▷.

- Execution of a Michelson program starts with a stack with one value, which is a pair (`param`, `st`) of a parameter `param` and a storage value `st`.
- `CDR` pops the pair at the top of the stack and pushes the second value of the popped pair; thus, after executing the instruction, the stack contains the single value `st`.
- `NIL` pushes the empty list [] to the stack; the instruction is accompanied by the type `operation` of the list elements for typechecking purposes.
- `AMOUNT` pushes the nonnegative `amount` of the money sent to the account to which this program is associated.
- `PUSH mutez 0` pushes the value 0. The type `mutez` represents a unit of money used in Tezos.
- `IFCMPEQ b1 b2`, if the state of the stack before executing the instruction is `v1` ▷ `v2` ▷ `tl`, (1) pops `v1` and `v2` and (2) executes the then-branch `b1` (resp., the else-branch `b2`) if `v2` = `v1` (resp., `v2` ≠ `v1`). In `boomerang`, this instruction does nothing if `amount` = 0; otherwise, the instructions in the else-branch are executed.
- `SOURCE` at the beginning of the else-branch pushes the address `src` of the source account, which initiated the chain of contract invocations that the current contract belongs to, resulting in the stack `src` ▷ [] ▷ `st`.
- `CONTRACT` *T* pops an address `addr` from the stack and typechecks whether the contract associated with `addr` takes an argument of type *T*. If the typechecking succeeds, then `Some (Contract addr)` is pushed; otherwise, `None` is pushed. The constructor `Contract` creates an object that represents a typechecked contract at the given address. In Tezos, the source account is always a contract that takes the value `Unit` as a parameter; thus, `Some (Contract src)` will always be pushed onto the stack.
- `ASSERT_SOME` pops a value `v` from the stack and pushes `v′` if `v` is `Some v′`; otherwise, it raises an exception.
- `UNIT` pushes the unit value `Unit` to the stack.
- `TRANSFER_TOKENS`, if the stack is of the shape `varg` ▷ `vamt` ▷ `vcontr` ▷ `tl`, pops `varg`, `vamt`, and `vcontr` from the stack and pushes `(Transfer varg vamt vcontr)` onto `tl`. The value `Transfer varg vamt vcontr` is an *operation object* expressing that money (of amount `vamt`) shall be sent to the account `vcontr` with the argument `varg` after this program finishes without rais-

ing an exception. Therefore, the program associated with `vcontr` is invoked after this program finishes. Otherwise, an operation object is an opaque tuple and no instruction can extract its elements.
- `CONS` with the stack `v1` ▷ `v2` ▷ `tl` pops `v1` and `v2`, and pushes a cons list `v1::v2` onto the stack. (We use the list notation in OCaml here.)
- After executing one of the branches associated with `IFCMPEQ` in this program, the shape of the stack should be `ops` ▷ `storage`, where `ops` is [] if `amount` = 0 or `[Transfer varg vamt vcontr]` if `amount` > 0. The instruction `PAIR` pops `ops` and `storage`, and pushes (`ops`, `storage`).

A Michelson program is supposed to finish its execution with a singleton stack whose unique element is a pair of (1) a list of operations to be executed after the current execution of the contract finishes and (2) the new value for the storage.

Michelson is a statically typed language. Each instruction is associated with a typing rule that specifies the shapes of stacks before and after it by a sequence of simple types such as `int` and `int list`. For example, `CONS` requires the type of top element to be T and that of the second to be T `list` (for any T); it ensures the top element after it has type T `list`.

Other notable features of Michelson include first-class functions, hashing, instructions related to cryptography such as signature verification, and manipulation of a blockchain using operations.

Specification

A user can specify the behavior of a program by a `ContractAnnot` annotation, which is a part of the augmented syntax of our verification tool. A `ContractAnnot` annotation gives a specification of a Michelson program by the following notation inspired by the refinement types: `{(param,st) | pre} -> {(ops,st') | post} & {exc | abpost}`, where `pre`, `post`, and `abpost` are predicates.

This specification reads as follows: if this program is invoked with a parameter `param` and storage `st` that satisfy the property `pre`, then (1) if the execution of this program succeeds, then it returns a list of operations `ops` and new storage `st′` that satisfy the property `post`; (2) if this program raises an exception with value `exc`, then `exc` satisfies `abpost`. The specification language, which is ML-like, is expressive enough to cover the specifications for practical contracts, including the ones we used in the experiments in 'Experiments'. In the predicates, one can use several keywords such as `amount` for the amount of the money sent to this program when it is invoked and `source` for the source account's address.

The `ContractAnnot` annotation in Fig. 1 (Lines 3–8) formalizes this program's specification as follows. This program can take any parameter and storage (Line 3). Successful execution of this program results in a pair (`ops`, `st′`) that satisfies the condition in Lines 4–7 that expresses (1) if `amount` = 0, then `ops` is empty, that is,

$$
\begin{aligned}
V &::= i \mid a \mid \texttt{Transfer}(V,i,a) \mid (V_1,V_2) \mid [] \mid V_1 :: V_2 \mid \langle IS \rangle \\
T &::= \texttt{int} \mid \texttt{address} \mid \texttt{operation} \mid T_1 \times T_2 \mid T\,\texttt{list} \mid T_1 \to T_2 \\
IS &::= \{\} \mid \{I; IS\} \\
I &::= \texttt{DIP}\,IS \mid \texttt{DROP} \mid \texttt{DUP} \mid \texttt{SWAP} \mid \texttt{PUSH}\,T\,V \mid \texttt{NOT} \mid \texttt{ADD} \mid \texttt{IF}\,IS_1\,IS_2 \mid \texttt{LOOP}\,IS \mid \texttt{PAIR} \mid \\
&\quad \texttt{CAR} \mid \texttt{CDR} \mid \texttt{NIL}\,T \mid \texttt{CONS} \mid \texttt{IF\_CONS}\,IS_1\,IS_2 \mid \texttt{ITER}\,IS \mid \texttt{LAMBDA}\,T_1\,T_2\,IS \mid \texttt{EXEC} \mid \\
&\quad \texttt{TRANSFER\_TOKENS}\,T \\
S &::= \ddagger \mid V \triangleright S \\
\bar{T} &::= \ddagger \mid T \triangleright \bar{T} \\
\Upsilon &::= \ddagger \mid x{:}T \triangleright \Upsilon
\end{aligned}
$$

Fig. 2 Syntax of Mini-Michelson

no operation will be issued; (2) if `amount` > 0, then `ops` is a list of a single element `Transfer Unit amount c`, where `c` is bound for `Contract source`[2], which expresses transfer of money of the amount `amount` to the account at `source` with the unit argument.[3] In the specification language, `source` and `amount` are keywords that stand for the source account and the amount of money sent to this program, respectively. The part & { _ | `False`} expresses that this program does not raise an exception. This specification correctly formalizes the intended behavior of this program.

Refinement Type System for Mini-Michelson

In this section, we formalize Mini-Michelson, a core subset of Michelson with its syntax, operational semantics, and refinement type system. We omit many features from the full language in favor of conciseness but includes language constructs—such as higher-order functions and iterations—that make verification difficult.

Syntax

Figure 2 shows the syntax of Mini-Michelson . *Values*, ranged over by V, consist of integers i; addresses a; operation objects $\texttt{Transfer}(V, i, a)$ to invoke a contract at a by sending money of amount i and an argument V; pairs (V_1, V_2) of values; the empty list []; cons $V_1 :: V_2$; and code $\langle IS \rangle$ of first-class functions.[4] Unlike Michelson, which has primitive Boolean literals `True` and `False`, we use integers as a substitute for Boolean values so that 0 means `False` and the others mean `True`. As we have mentioned, there is no instruction to extract elements from an operation object but the elements can be referenced in refinement types to state what kind of operation object is constructed by a smart contract. *Simple types*, ranged

[2] It is one axiom of our domain specific theory that `contract_opt source` always return `Some (Contract source)`.

[3] As we mentioned in 'Introduction', Helmholtz can currently verify the behavior of a single contract, although there will be an invocation of the contract associated with `source` after the termination of `boomerang`. An operation is treated as an opaque data structure, from which one cannot extract values.

[4] Closures are not needed because functions in Michelson can access only arguments.

over by T, consist of base types (`int`, `address`, and `operation`, which are self-explanatory), pair types $T_1 \times T_2$, list types T `list`, and function types $T_1 \to T_2$. *Instruction sequences*, ranged over by *IS*, are a sequence of *instructions*, ranged over by I, enclosed by curly braces. A Mini-Michelson *program* is an instruction sequence.

Instructions include those for operand stack manipulation (to `DROP`, `DUP`licate, `SWAP`, and `PUSH` values); `NOT` and `ADD` for manipulating integers; `PAIR`, `CAR`, and `CDR` for pairs; `NIL` and `CONS` for constructing lists; `LAMBDA` for a first-class function; `EXEC` for calling a function; and `TRANSFER_TOKENS` to create an operation. Instructions for control structures are `IF` and `IF_CONS`, which are for branching on integers (whether the stack top is `True` or not) and lists (whether the stack top is a cons or not), respectively, and `LOOP` and `ITER`, which are for iteration on integers and lists, respectively. `LAMBDA` pushes a function (described by its operand *IS*) onto the stack and `EXEC` calls a function. Perhaps unfamiliar is `DIP` *IS*, which pops and saves the stack top somewhere else, executes *IS*, and then pushes back the saved value.

We also use a few kinds of stacks in the following definitions: operand stacks, ranged over by S, type stacks, ranged over by $\bar{T}$, and type binding stacks, ranged over by Y. The empty stack is denoted by $\ddagger$ and push is by $\rhd$. We often omit the empty stack and write, for example, $V_1 \rhd V_2$ for $V_1 \rhd V_2 \rhd \ddagger$. Intuitively, $T_1 \rhd .. \rhd T_n$ and $x_1 : T_1 \rhd .. \rhd x_n : T_n$ describe stacks $V_1 \rhd .. \rhd V_n$, where each value V_i is of type T_i. We will use variables to name stack elements in the refinement type system.

We summarize main differences from Michelson proper:

- Michelson has the notion of type attributes, which classify types, according to which generic operations such as `PUSH` can be applied. For example, values of *pushable* types can be put on the stack by `PUSH`. Since type `operation` is not pushable, an instruction such as `PUSH operation Transfer`(V, i, a) is not valid in Michelson—all operations have to be created by designated instructions. We ignore type attributes for simplicity here, but the implementation of Helmholtz, which calls the typechecker of Michelson, does not.
- As we saw in 'Overview of HELMHOLTZ and Michelson', an operation is created from an address in two steps via a contract value. Since we model only one kind of operations, i.e., `Transfer`(V, i, a), we simplify the process to let instruction `TRANSFER_TOKENS` directly creates an operation from an address in one step. We also omit the typecheck of the contract associated with an address.
- In Michelson, each execution of a smart contract is assigned a *gas* to control how long the contract can run to prevent contracts from running too long. Current Helmholtz, however, never takes into account gas consumption because it depends on the *size* of values manipulated. Incorporating the gas consumption in our framework is left as future work.
- We do not formally model exceptions for simplicity and, thus, the refinement type system does not (have to) capture exceptional behavior. Our verifier, however, *does* handle exceptions; we will informally discuss how we extend the type system with exceptions in Section 'Extension with Exceptions'.

Evaluation for Instruction Sequence $\boxed{S \vdash IS \Downarrow S}$

$$\frac{}{S \vdash \{\} \Downarrow S}\ (\text{E-Nop}) \qquad \frac{S \vdash I \Downarrow S' \qquad S' \vdash IS \Downarrow S''}{S \vdash \{I; IS\} \Downarrow S''}\ (\text{E-Seq})$$

Evaluation for Instruction $\boxed{S \vdash I \Downarrow S}$

$$\frac{S \vdash IS \Downarrow S'}{V \rhd S \vdash \texttt{DIP}\, IS \Downarrow V \rhd S'}\ (\text{E-Dip}) \qquad \frac{}{V \rhd S \vdash \texttt{DROP} \Downarrow S}\ (\text{E-Drop}) \qquad \frac{}{V \rhd S \vdash \texttt{DUP} \Downarrow V \rhd V \rhd S}\ (\text{E-Dup})$$

$$\frac{}{V_1 \rhd V_2 \rhd S \vdash \texttt{SWAP} \Downarrow V_2 \rhd V_1 \rhd S}\ (\text{E-Swap}) \qquad \frac{}{S \vdash \texttt{PUSH}\, T\, V \Downarrow V \rhd S}\ (\text{E-Push})$$

$$\frac{(i \neq 0)}{i \rhd S \vdash \texttt{NOT} \Downarrow 0 \rhd S}\ (\text{E-NotT}) \qquad \frac{}{0 \rhd S \vdash \texttt{NOT} \Downarrow 1 \rhd S}\ (\text{E-NotF}) \qquad \frac{(i_1 + i_2 = i_3)}{i_1 \rhd i_2 \rhd S \vdash \texttt{ADD} \Downarrow i_3 \rhd S}\ (\text{E-Add})$$

$$\frac{}{V_1 \rhd V_2 \rhd S \vdash \texttt{PAIR} \Downarrow (V_1, V_2) \rhd S}\ (\text{E-Pair}) \qquad \frac{}{(V_1, V_2) \rhd S \vdash \texttt{CAR} \Downarrow V_1 \rhd S}\ (\text{E-CAR})$$

$$\frac{}{(V_1, V_2) \rhd S \vdash \texttt{CDR} \Downarrow V_2 \rhd S}\ (\text{E-CDR}) \qquad \frac{}{S \vdash \texttt{NIL}\, T \Downarrow [] \rhd S}\ (\text{E-Nil})$$

$$\frac{}{V_1 \rhd V_2 \rhd S \vdash \texttt{CONS} \Downarrow V_1 :: V_2 \rhd S}\ (\text{E-Cons}) \qquad \frac{(i \neq 0) \qquad S \vdash IS_1 \Downarrow S'}{i \rhd S \vdash \texttt{IF}\, IS_1\, IS_2 \Downarrow S'}\ (\text{E-IfT})$$

$$\frac{S \vdash IS_2 \Downarrow S'}{0 \rhd S \vdash \texttt{IF}\, IS_1\, IS_2 \Downarrow S'}\ (\text{E-IfF}) \qquad \frac{(i \neq 0) \qquad S \vdash IS \Downarrow S' \qquad S' \vdash \texttt{LOOP}\, IS \Downarrow S''}{i \rhd S \vdash \texttt{LOOP}\, IS \Downarrow S''}\ (\text{E-LoopT})$$

$$\frac{}{0 \rhd S \vdash \texttt{LOOP}\, IS \Downarrow S}\ (\text{E-LoopF}) \qquad \frac{V_1 \rhd V_2 \rhd S \vdash IS_1 \Downarrow S'}{V_1 :: V_2 \rhd S \vdash \texttt{IF\_CONS}\, IS_1\, IS_2 \Downarrow S'}\ (\text{E-IfConsT})$$

$$\frac{S \vdash IS_2 \Downarrow S'}{[] \rhd S \vdash \texttt{IF\_CONS}\, IS_1\, IS_2 \Downarrow S'}\ (\text{E-IfConsF}) \qquad \frac{}{[] \rhd S \vdash \texttt{ITER}\, IS \Downarrow S}\ (\text{E-IterNil})$$

$$\frac{V_1 \rhd S \vdash IS \Downarrow S' \qquad V_2 \rhd S' \vdash \texttt{ITER}\, IS \Downarrow S''}{V_1 :: V_2 \rhd S \vdash \texttt{ITER}\, IS \Downarrow S''}\ (\text{E-IterCons}) \qquad \frac{}{S \vdash \texttt{LAMBDA}\, T_1\, T_2\, IS \Downarrow \langle IS \rangle \rhd S}\ (\text{E-Lambda})$$

$$\frac{V \rhd \ddagger \vdash IS \Downarrow V' \rhd \ddagger}{V \rhd \langle IS \rangle \rhd S \vdash \texttt{EXEC} \Downarrow V' \rhd S}\ (\text{E-Exec})$$

$$\frac{}{V \rhd i \rhd a \rhd S \vdash \texttt{TRANSFER\_TOKENS}\, T \Downarrow \texttt{Transfer}(V, i, a) \rhd S}\ (\text{E-TransferTokens})$$

Fig. 3 Operational semantics of Mini-Michelson

Operational Semantics

Figure 3 defines the operational semantics of Mini-Michelson . A judgment of the form $S \vdash I \Downarrow S'$ (or $S \vdash IS \Downarrow S'$, resp.) means that evaluating the instruction I (or the instruction sequence IS, resp.) under the stack S results in the stack S'.

Although the defining rules are straightforward, we will make a few remarks about them.

The rule (E-Dip) means that DIP *IS* pops and saves the stack top somewhere else, executes *IS*, and then push back the saved value, as explained above. This instruction implicitly gives Mini-Michelson (and Michelson) a secondary stack. (E-Push) means that PUSH *T V* does not check if the pushed value is well formed at run time: the check is the job of the simple type system, discussed soon.

The rules (E-IfT) and (E-IfF) define the behavior of the branching instruction IF IS_1 IS_2, which executes IS_1 or IS_2, depending on the top of the operand stack. As we have mentioned, nonzero integers mean True. Thus, (E-IfT) is used for the case in which IS_1 is executed, and otherwise, (E-IfF) is used. There is another branching instruction IF_CONS IS_1 IS_2, which executes either instruction sequence depending on whether the list at the top of the stack is empty or not (cf. (E-IfConsT) and (E-IfConsF)).

The rules (E-LoopT) and (E-LoopF) define the behavior of the looping instruction LOOP *IS*. This instruction executes *IS* repeatedly until the top of the stack becomes False. (E-LoopT) means that, if the condition is True, *IS* is executed, and then LOOP *IS* is executed again. (E-LoopF) means that, if the condition is False, the loop is finished after dropping the stack top. A similar looping instruction is ITER *IS*, which iterates over a list (see (E-IterNil) and (E-IterCons)).

The rule (E-Lambda) means that LAMBDA T_1 T_2 *IS* pushes the instruction sequence to the stack and (E-Exec) means that EXEC pops the instruction sequence $\langle IS \rangle$ and the stack top V, saves the rest of the stack S elsewhere, runs *IS* with V as the sole value in the stack, pushes the result V' back to the restored stack S.

The rule (E-TransferTokens) means that TRANSFER_TOKENS *T* creates an operation object and pushes onto the stack. (As we have discussed, we omit a run-time check to see if *T* is really the argument type of the contract that the address *a* stores.)

Simple Type System

Mini-Michelson (as well as Michelson) is equipped with a simple type system. The type judgment for instructions is written $\bar{T} \vdash I \Rightarrow \bar{T}'$, which means that instruction I transforms a stack of type $\bar{T}$ into another stack of type $\bar{T}'$. The type judgment for values is written $V : T$, which means that V is given simple type T. The typing rules, which are shown in Fig. 4, are fairly straightforward. Note that these two judgment forms depend on each other—see (RTV-Fun) and (T-Push).

Refinement Type System

Now, we extend the simple type system to a refinement type system. In the refinement type system, a simple stack type $T_1 \triangleright .. \triangleright T_n$ is augmented with a formula φ in an assertion language to describe the relationship among stack elements. More concretely, we introduce *refinement stack types*, ranged over by Φ, of the form $\{x_1 : T_1 \triangleright ... \triangleright x_n : T_n \mid \varphi(x_1, ..., x_n)\}$, which denotes a stack $V_1 \triangleright .. \triangleright V_n$ such that $V_1 : T_1$, ..., $V_n : T_n$ and $\varphi(V_1, ..., V_n)$ hold, and refine the type judgment form,

Value Typing $\boxed{V : T}$

$$\frac{}{i : \texttt{int}}\ (\text{RTV-INT}) \qquad \frac{}{a : \texttt{address}}\ (\text{RTV-ADDRESS})$$

$$\frac{V : T}{\texttt{Transfer}(V, i, a) : \texttt{operation}}\ (\text{RTV-OPERATION}) \qquad \frac{V_1 : T_1 \qquad V_2 : T_2}{(V_1, V_2) : T_1 \times T_2}\ (\text{RTV-PAIR})$$

$$\frac{}{[] : T\ \texttt{list}}\ (\text{RTV-NIL}) \quad \frac{V_1 : T \qquad V_2 : T\ \texttt{list}}{V_1 :: V_2 : T\ \texttt{list}}\ (\text{RTV-CONS}) \quad \frac{T_1 \triangleright \ddagger \vdash IS \Rightarrow T_2 \triangleright \ddagger}{\langle IS \rangle : T_1 \to T_2}\ (\text{RTV-FUN})$$

Instruction Sequence Typing $\boxed{\bar{T} \vdash IS \Rightarrow \bar{T}'}$

$$\frac{}{\bar{T} \vdash \{\} \Rightarrow \bar{T}}\ (\text{T-NOP}) \qquad \frac{\bar{T}_1 \vdash I \Rightarrow \bar{T}_2 \qquad \bar{T}_2 \vdash IS \Rightarrow \bar{T}_3}{\bar{T}_1 \vdash \{I; IS\} \Rightarrow \bar{T}_3}\ (\text{T-SEQ})$$

Instruction Typing $\boxed{\bar{T} \vdash I \Rightarrow \bar{T}'}$

$$\frac{\bar{T}_1 \vdash IS \Rightarrow \bar{T}_2}{T \triangleright \bar{T}_1 \vdash \texttt{DIP}\, IS \Rightarrow T \triangleright \bar{T}_2}\ (\text{T-DIP}) \quad \frac{}{T \triangleright \bar{T} \vdash \texttt{DROP} \Rightarrow \bar{T}}\ (\text{T-DROP}) \quad \frac{}{T \triangleright \bar{T} \vdash \texttt{DUP} \Rightarrow T \triangleright T \triangleright \bar{T}}\ (\text{T-DUP})$$

$$\frac{}{T_1 \triangleright T_2 \triangleright \bar{T} \vdash \texttt{SWAP} \Rightarrow T_2 \triangleright T_1 \triangleright \bar{T}}\ (\text{T-SWAP}) \qquad \frac{V : T}{\bar{T} \vdash \texttt{PUSH}\, T\, V \Rightarrow T \triangleright \bar{T}}\ (\text{T-PUSH})$$

$$\frac{}{\texttt{int} \triangleright \bar{T} \vdash \texttt{NOT} \Rightarrow \texttt{int} \triangleright \bar{T}}\ (\text{T-NOT}) \qquad \frac{}{\texttt{int} \triangleright \texttt{int} \triangleright \bar{T} \vdash \texttt{ADD} \Rightarrow \texttt{int} \triangleright \bar{T}}\ (\text{T-ADD})$$

$$\frac{}{T_1 \triangleright T_2 \triangleright \bar{T} \vdash \texttt{PAIR} \Rightarrow T_1 \times T_2 \triangleright \bar{T}}\ (\text{T-PAIR}) \qquad \frac{}{T_1 \times T_2 \triangleright \bar{T} \vdash \texttt{CAR} \Rightarrow T_1 \triangleright \bar{T}}\ (\text{T-CAR})$$

$$\frac{}{T_1 \times T_2 \triangleright \bar{T} \vdash \texttt{CDR} \Rightarrow T_2 \triangleright \bar{T}}\ (\text{T-CDR}) \qquad \frac{}{\bar{T} \vdash \texttt{NIL}\, T \Rightarrow T\ \texttt{list} \triangleright \bar{T}}\ (\text{T-NIL})$$

$$\frac{}{T \triangleright T\ \texttt{list} \triangleright \bar{T} \vdash \texttt{CONS} \Rightarrow T\ \texttt{list} \triangleright \bar{T}}\ (\text{T-CONS}) \qquad \frac{\bar{T} \vdash IS_1 \Rightarrow \bar{T}' \qquad \bar{T} \vdash IS_2 \Rightarrow \bar{T}'}{\texttt{int} \triangleright \bar{T} \vdash \texttt{IF}\, IS_1\, IS_2 \Rightarrow \bar{T}'}\ (\text{T-IF})$$

$$\frac{\bar{T} \vdash IS \Rightarrow \texttt{int} \triangleright \bar{T}}{\texttt{int} \triangleright \bar{T} \vdash \texttt{LOOP}\, IS \Rightarrow \bar{T}}\ (\text{T-LOOP}) \qquad \frac{T \triangleright T\ \texttt{list} \triangleright \bar{T} \vdash IS_1 \Rightarrow \bar{T}' \qquad \bar{T} \vdash IS_2 \Rightarrow \bar{T}'}{T\ \texttt{list} \triangleright \bar{T} \vdash \texttt{IF\_CONS}\, IS_1\, IS_2 \Rightarrow \bar{T}'}\ (\text{T-IFCONS})$$

$$\frac{T \triangleright \bar{T} \vdash IS \Rightarrow \bar{T}}{T\ \texttt{list} \triangleright \bar{T} \vdash \texttt{ITER}\, IS \Rightarrow \bar{T}}\ (\text{T-ITER}) \qquad \frac{T_1 \triangleright \ddagger \vdash IS \Rightarrow T_2 \triangleright \ddagger}{\bar{T} \vdash \texttt{LAMBDA}\, T_1\, T_2\, IS \Rightarrow T_1 \to T_2 \triangleright \bar{T}}\ (\text{T-LAMBDA})$$

$$\frac{}{T_1 \triangleright T_1 \to T_2 \triangleright \bar{T} \vdash \texttt{EXEC} \Rightarrow T_2 \triangleright \bar{T}}\ (\text{T-EXEC})$$

$$\frac{}{T \triangleright \texttt{int} \triangleright \texttt{address} \triangleright \bar{T} \vdash \texttt{TRANSFER\_TOKENS}\, T \Rightarrow \texttt{operation} \triangleright \bar{T}}\ (\text{T-TRANSFERTOKENS})$$

Fig. 4 Simple typing

$$
\begin{aligned}
t &::= x \mid i \mid a \mid [] \mid \langle IS \rangle \mid \texttt{Transfer}(t_1, t_2, t_3) \mid (t_1, t_2) \mid t_1 :: t_2 \mid t_1 + t_2 \\
\varphi &::= \top \mid t_1 = t_2 \mid \texttt{call}(t_1, t_2) = t_3 \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \exists x{:}T.\varphi \\
\Gamma &::= \texttt{empty} \mid \Gamma, x{:}T
\end{aligned}
$$

Fig. 5 Syntax of Assertion Language

accordingly. We start with an assertion language, which is many-sorted first-order logic and proceed to the refinement type system.

Assertion Language

The assertion language is many-sorted first-order logic, where sorts are simple types. We show the syntax of terms, ranged over by t, and formulae, ranged over by φ, in Fig. 5. As usual, x is bound in $\exists x : T.\varphi$. Most of them are straightforward but some constructions are worth explaining. A formula of the form $\texttt{call}(t_1, t_2) = t_3$ means that, if instruction sequence denoted by t_1 is called with (a singleton stack that stores) a value denoted by t_2 (and terminates), it yields the value denoted by t_3. The term constructor $\texttt{Transfer}(t_1, t_2, t_3)$ allows us to refer to the elements in an operation object, which is opaque. Conjunction $\varphi_1 \wedge \varphi_2$, implication $\varphi_1 \implies \varphi_2$, and universal quantification $\forall x : T.\varphi$ are defined as abbreviations as usual. We use several common abbreviations such as $t_1 \neq t_2$ for $\neg(t_1 = t_2)$, $\exists x_1 : T_1, .., x_n : T_n.\varphi$ for $\exists x_1 : T_1. \ldots \exists x_n : T_n.\varphi$, etc. A typing environment, ranged over by Γ, is a sequence of type binding. We assume all variables in Γ are distinct. We abuse a comma to concatenate typing environments, e.g., Γ_1, Γ_2. We also use a type binding stack Y as a typing environment, explicitly denoted by $\widehat{Y}$, which is defined as $\widehat{\ddagger} = \texttt{empty}$ and $\widehat{x : T \rhd Y} = x : T, \widehat{Y}$. To save space, we (ab) use V to denote a subset of terms and the value typing (see (WT-Val)). Strictly speaking, a term like $(0, 0)$ has two type derivations but it is easy to show that a term has at most one type under a given type environment. A similar abuse of V will be found elsewhere (such as Definition 2 below).

Well-sorted terms and formulae are defined by the judgments $\Gamma \vdash t : T$ and $\Gamma \vdash \varphi : *$, respectively. The former means that the term t is a well-sorted term of the sort T under the typing environment Γ and the latter that the formula φ is well sorted under the typing environment Γ, respectively. The derivation rules for each judgment, shown in Fig. 6, are straightforward. Note that well-sortedness depends on the simple type system via (WT-Val).

Let a *value assignment* σ be a mapping from variables to values. We write $\sigma[x \mapsto V]$ to denote the value assignment which maps x to V and otherwise is identical to σ. As we are interested in well-sorted formulae, we consider a value assignment that respects a typing environment, as follows.

Definition 1 A value assignment σ is *typed* under a typing environment Γ, denoted by $\sigma : \Gamma$, iff $\sigma(x) : T$ for every $x : T \in \Gamma$.

Well-Sorted Terms $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T}\ (\text{WT-VAR}) \qquad \frac{V : T}{\Gamma \vdash V : T}\ (\text{WT-VAL}) \qquad \frac{\Gamma \vdash t_1 : \texttt{int} \qquad \Gamma \vdash t_2 : \texttt{int}}{\Gamma \vdash t_1 + t_2 : \texttt{int}}\ (\text{WT-PLUS})$$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : \texttt{int} \qquad \Gamma \vdash t_3 : \texttt{address}}{\Gamma \vdash \texttt{Transfer}(t_1, t_2, t_3) : \texttt{operation}}\ (\text{WT-TRANSACTION})$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 \times T_2}\ (\text{WT-PAIR}) \qquad \frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T\ \texttt{list}}{\Gamma \vdash t_1 :: t_2 : T\ \texttt{list}}\ (\text{WT-CONS})$$

Well-Sorted Formulae $\boxed{\Gamma \vdash \varphi : *}$

$$\frac{}{\Gamma \vdash \top : *}\ (\text{WF-TRUE}) \qquad \frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 = t_2 : *}\ (\text{WF-EQUAL})$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1 \qquad \Gamma \vdash t_3 : T_2}{\Gamma \vdash \texttt{call}(t_1, t_2) = t_3 : *}\ (\text{WF-CALL}) \qquad \frac{\Gamma \vdash \varphi : *}{\Gamma \vdash \neg \varphi : *}\ (\text{WF-NOT})$$

$$\frac{\Gamma \vdash \varphi_1 : * \qquad \Gamma \vdash \varphi_2 : *}{\Gamma \vdash \varphi_1 \vee \varphi_2 : *}\ (\text{WF-OR}) \qquad \frac{\Gamma, x{:}T \vdash \varphi : *}{\Gamma \vdash \exists x{:}T.\varphi : *}\ (\text{WF-EXISTS})$$

Fig. 6 Well-Sorted Terms and Formulae

Now, we define the semantics of the well-sorted terms and formulae in a standard manner as follows.

Definition 2 (Semantics of terms) The semantics $[\![t]\!]_{\sigma:\Gamma}$ of term t under typed value assignment $\sigma : \Gamma$ is defined as follows:

$$\begin{aligned}
[\![x]\!]_{\sigma:\Gamma} &= \sigma(x) \qquad [\![V]\!]_{\sigma:\Gamma} = V \\
[\![\texttt{Transfer}(t_1, t_2, t_3)]\!]_{\sigma:\Gamma} &= \texttt{Transfer}([\![t_1]\!]_{\sigma:\Gamma}, [\![t_2]\!]_{\sigma:\Gamma}, [\![t_3]\!]_{\sigma:\Gamma}) \\
[\![(t_1, t_2)]\!]_{\sigma:\Gamma} &= ([\![t_1]\!]_{\sigma:\Gamma}, [\![t_2]\!]_{\sigma:\Gamma}) \\
[\![t_1 :: t_2]\!]_{\sigma:\Gamma} &= [\![t_1]\!]_{\sigma:\Gamma} :: [\![t_2]\!]_{\sigma:\Gamma} \\
[\![t_1 + t_2]\!]_{\sigma:\Gamma} &= [\![t_1]\!]_{\sigma:\Gamma} + [\![t_2]\!]_{\sigma:\Gamma}.
\end{aligned}$$

Definition 3 (Semantics of formulae). For a typed value assignment $\sigma : \Gamma$, a *valid* well-sorted formula φ under Γ is denoted by $\sigma : \Gamma \vDash \varphi$ and defined as follows.

- $\sigma : \Gamma \vDash \top$.
- $\sigma : \Gamma \vDash t_1 = t_2$ iff $[\![t_1]\!]_{\sigma:\Gamma} = [\![t_2]\!]_{\sigma:\Gamma}$.
- $\sigma : \Gamma \vDash \texttt{call}(t_1, t_2) = t_3$ iff $[\![t_1]\!]_{\sigma:\Gamma} = \langle IS \rangle$ and $[\![t_2]\!]_{\sigma:\Gamma} \rhd \ddagger \vdash IS \Downarrow [\![t_3]\!]_{\sigma:\Gamma} \rhd \ddagger$.
- $\sigma : \Gamma \vDash \neg \varphi$ iff $\sigma : \Gamma \nvDash \varphi$.
- $\sigma : \Gamma \vDash \varphi_1 \vee \varphi_2$ iff $\sigma : \Gamma \vDash \varphi_1$ or $\sigma : \Gamma \vDash \varphi_2$.
- $\sigma : \Gamma \vDash \exists x : T.\varphi$ iff $\sigma[x \mapsto V] : \Gamma, x : T \vDash \varphi$ for some V.

We write $\Gamma \vDash \varphi$ iff $\sigma : \Gamma \vDash \varphi$ for any σ.

Typing Rules

The type system is defined by subtyping and typing: a subtyping judgment is of the form $\Gamma \vdash \Phi_1 <: \Phi_2$, which means stack type Φ_1 is a subtype of Φ_2 under Γ, and a type judgment for instructions (resp. instruction sequences) is of the form $\Gamma \vdash \Phi_1\ I\ \Phi_2$ (resp. $\Gamma \vdash \Phi_1\ IS\ \Phi_2$), which means that if I (resp. IS) is executed under a stack satisfying Φ_1, the resulting stack (if terminates) satisfies Φ_2. We often call Φ_1 *pre-condition* and Φ_2 *post-condition*, following the terminology of Hoare logic. Note that the scopes of variables declared in Γ include Φ_1 and Φ_2 but those bound in Φ_1 *do not* include Φ_2. To express the relationship between the initial and final stacks, we use the type environment Γ. In writing down concrete specifications, it is sometimes convenient to allow the scopes of variables bound in Φ_1 to include Φ_2, but we find that it would clutter the presentation of typing rules.

In our type system, subtyping is defined semantically as follows.

Definition 4 (Subtyping relation). A refinement stack type $\{Y \mid \varphi_1\}$ is called *subtype* of a refinement stack type $\{Y \mid \varphi_2\}$ under a typing environment Γ, denoted by $\Gamma \vdash \{Y \mid \varphi_1\} <: \{Y \mid \varphi_2\}$, iff $\Gamma, \widehat{Y} \vDash \varphi_1 \implies \varphi_2$.

We show the typing rules in Figs. 7 and 8. It is easy to observe that the type binding stack parts in the pre- and post-conditions follows the simple type system. We will focus on predicate parts below.

- (RT-Dip) means that `DIP` *IS* is well typed if the body *IS* is typed under the stack type obtained by removing the top element. However, since a property φ for the initial stack relies on the popped value x, we keep the binding in the typing environment.
- (RT-If) means that the instruction is well typed if both branches have the same post-condition; the pre-conditions of the branches are strengthened by the assumptions that the top of the input stack is `True` ($x \neq 0$) and `False` ($x = 0$). The variable x is existentially quantified because the top element will be removed before the execution of either branch.
- (RT-Loop) is similar to the proof rule for while-loops in Hoare logic. The formula φ is a loop invariant. Since the body of `LOOP` is executed while the stack top is nonzero, the pre-condition for the body *IS* is strengthened by $x \neq 0$, whereas the post-condition of `LOOP` *IS* is strengthened by $x = 0$.
- (RT-Iter) can be understood as a variant of (RT-Loop). In the premise, $x_1 :: x_2 = x$ represents the condition under which iteration goes on, that is a list on top of the stack is non-nil. In addition to that, $x_2 = x$ guarantees that the loop-invariant φ holds for the tail of the list since in the next iteration stack top becomes the tail of the list.

Refinement Instruction Sequence Typing $\boxed{\Gamma \vdash \Phi_1\ IS\ \Phi_2}$

$$\frac{}{\Gamma \vdash \Phi\ \{\}\ \Phi}\ \text{(RT-NOP)} \qquad \frac{\Gamma \vdash \Phi\ I\ \Phi' \qquad \Gamma \vdash \Phi'\ IS\ \Phi''}{\Gamma \vdash \Phi\ \{I;IS\}\ \Phi''}\ \text{(RT-SEQ)}$$

Refinement Instruction Typing $\boxed{\Gamma \vdash \Phi_1\ I\ \Phi_2}$

$$\frac{\Gamma, x{:}T \vdash \{\Upsilon \mid \varphi\}\ IS\ \{\Upsilon' \mid \varphi'\}}{\Gamma \vdash \{x{:}T \triangleright \Upsilon \mid \varphi\}\ \texttt{DIP}\,IS\ \{x{:}T \triangleright \Upsilon' \mid \varphi'\}}\ \text{(RT-DIP)} \qquad \frac{}{\Gamma \vdash \{x{:}T \triangleright \Upsilon \mid \varphi\}\ \texttt{DROP}\ \{\Upsilon \mid \exists x{:}T.\varphi\}}\ \text{(RT-DROP)}$$

$$\frac{(x' \notin \mathrm{dom}(\Gamma, \widehat{x{:}T \triangleright \Upsilon}))}{\Gamma \vdash \{x{:}T \triangleright \Upsilon \mid \varphi\}\ \texttt{DUP}\ \{x'{:}T \triangleright x{:}T \triangleright \Upsilon \mid \varphi \wedge x' = x\}}\ \text{(RT-DUP)}$$

$$\frac{}{\Gamma \vdash \{x_1{:}T_1 \triangleright x_2{:}T_2 \triangleright \Upsilon \mid \varphi\}\ \texttt{SWAP}\ \{x_2{:}T_2 \triangleright x_1{:}T_1 \triangleright \Upsilon \mid \varphi\}}\ \text{(RT-SWAP)}$$

$$\frac{(x \notin \mathrm{dom}(\Gamma, \widehat{\Upsilon})) \qquad V : T}{\Gamma \vdash \{\Upsilon \mid \varphi\}\ \texttt{PUSH}\,T\,V\ \{x{:}T \triangleright \Upsilon \mid \varphi \wedge x = V\}}\ \text{(RT-PUSH)}$$

$$\frac{(x' \notin \mathrm{dom}(\Gamma, \widehat{x{:}\texttt{int} \triangleright \Upsilon}))}{\Gamma \vdash \{x{:}\texttt{int} \triangleright \Upsilon \mid \varphi\}\ \texttt{NOT}\ \{x'{:}\texttt{int} \triangleright \Upsilon \mid \exists x{:}\texttt{int}.\varphi \wedge (x \neq 0 \wedge x' = 0 \vee x = 0 \wedge x' = 1)\}}\ \text{(RT-NOT)}$$

$$\frac{(x_3 \notin \mathrm{dom}(\Gamma, x_1{:}\widehat{\texttt{int} \triangleright x_2{:}\texttt{int}} \triangleright \Upsilon))}{\Gamma \vdash \{x_1{:}\texttt{int} \triangleright x_2{:}\texttt{int} \triangleright \Upsilon \mid \varphi\}\ \texttt{ADD}\ \{x_3{:}\texttt{int} \triangleright \Upsilon \mid \exists x_1{:}\texttt{int}, x_2{:}\texttt{int}.\varphi \wedge x_1 + x_2 = x_3\}}\ \text{(RT-ADD)}$$

$$\frac{(x_3 \notin \mathrm{dom}(\Gamma, x_1{:}T_1 \widehat{\triangleright x_2{:}T_2} \triangleright \Upsilon))}{\Gamma \vdash \{x_1{:}T_1 \triangleright x_2{:}T_2 \triangleright \Upsilon \mid \varphi\}\ \texttt{PAIR}\ \{x_3{:}T_1 \times T_2 \triangleright \Upsilon \mid \exists x_1{:}T_1, x_2{:}T_2.\varphi \wedge (x_1, x_2) = x_3\}}\ \text{(RT-PAIR)}$$

$$\frac{(x_1 \neq x_2) \qquad (x_1 \notin \mathrm{dom}(\Gamma, \widehat{x{:}T_1 \times T_2} \triangleright \Upsilon)) \qquad (x_2 \notin \mathrm{dom}(\Gamma, \widehat{x{:}T_1 \times T_2} \triangleright \Upsilon))}{\Gamma \vdash \{x{:}T_1 \times T_2 \triangleright \Upsilon \mid \varphi\}\ \texttt{CAR}\ \{x_1{:}T_1 \triangleright \Upsilon \mid \exists x{:}T_1 \times T_2, x_2{:}T_2.\varphi \wedge x = (x_1, x_2)\}}\ \text{(RT-CAR)}$$

$$\frac{(x_1 \neq x_2) \qquad (x_1 \notin \mathrm{dom}(\Gamma, \widehat{x{:}T_1 \times T_2} \triangleright \Upsilon)) \qquad (x_2 \notin \mathrm{dom}(\Gamma, \widehat{x{:}T_1 \times T_2} \triangleright \Upsilon))}{\Gamma \vdash \{x{:}T_1 \times T_2 \triangleright \Upsilon \mid \varphi\}\ \texttt{CDR}\ \{x_2{:}T_2 \triangleright \Upsilon \mid \exists x{:}T_1 \times T_2, x_1{:}T_1.\varphi \wedge x = (x_1, x_2)\}}\ \text{(RT-CDR)}$$

$$\frac{(x \notin \mathrm{dom}(\Gamma, \widehat{\Upsilon}))}{\Gamma \vdash \{\Upsilon \mid \varphi\}\ \texttt{NIL}\,T\ \{x{:}T\,\texttt{list} \triangleright \Upsilon \mid \varphi \wedge x = []\}}\ \text{(RT-NIL)}$$

$$\frac{(x_3 \notin \mathrm{dom}(\Gamma, x_1{:}T \triangleright \widehat{x_2{:}T\,\texttt{list}} \triangleright \Upsilon))}{\Gamma \vdash \{x_1{:}T \triangleright x_2{:}T\,\texttt{list} \triangleright \Upsilon \mid \varphi\}\ \texttt{CONS}\ \{x_3{:}T\,\texttt{list} \triangleright \Upsilon \mid \exists x_1{:}T, x_2{:}T\,\texttt{list}.\varphi \wedge x_1 :: x_2 = x_3\}}\ \text{(RT-CONS)}$$

Fig. 7 Typing rules (I)

– (RT-LAMBDA) is for the instruction to push a first-class function onto the operand stack. The premise of the rule means that the body IS takes a value (named y_1) of type T_1 that satisfies φ_1 and outputs a value (named y_2) of type T_2 that satisfies φ_2

$$\frac{\Gamma \vdash \{\Upsilon \mid \exists x{:}\texttt{int}.\varphi \wedge x \neq 0\}\ IS_1\ \Phi \qquad \Gamma \vdash \{\Upsilon \mid \exists x{:}\texttt{int}.\varphi \wedge x = 0\}\ IS_2\ \Phi}{\Gamma \vdash \{x{:}\texttt{int} \triangleright \Upsilon \mid \varphi\}\ \texttt{IF}\, IS_1\, IS_2\ \Phi} \text{ (RT-IF)}$$

$$\frac{\Gamma \vdash \{\Upsilon \mid \exists x{:}\texttt{int}.\varphi \wedge x \neq 0\}\ IS\ \{x{:}\texttt{int} \triangleright \Upsilon \mid \varphi\}}{\Gamma \vdash \{x{:}\texttt{int} \triangleright \Upsilon \mid \varphi\}\ \texttt{LOOP}\, IS\ \{\Upsilon \mid \exists x{:}\texttt{int}.\varphi \wedge x = 0\}} \text{ (RT-LOOP)}$$

$$\frac{\begin{array}{l}(x_1 \neq x_2) \quad (x_1 \notin \mathrm{dom}(\Gamma, x{:}T\,\widehat{\texttt{list}} \triangleright \Upsilon)) \quad (x_2 \notin \mathrm{dom}(\Gamma, x{:}T\,\widehat{\texttt{list}} \triangleright \Upsilon)) \\ \Gamma \vdash \{x_1{:}T \triangleright x_2{:}T\,\texttt{list} \triangleright \Upsilon \mid \exists x{:}T\,\texttt{list}.\varphi \wedge x_1 :: x_2 = x\}\ IS_1\ \Phi \\ \Gamma \vdash \{\Upsilon \mid \exists x{:}T\,\texttt{list}.\varphi \wedge x = []\}\ IS_2\ \Phi\end{array}}{\Gamma \vdash \{x{:}T\,\texttt{list} \triangleright \Upsilon \mid \varphi\}\ \texttt{IF\_CONS}\, IS_1\, IS_2\ \Phi} \text{ (RT-IFCONS)}$$

$$\frac{\begin{array}{l}(x_1 \neq x_2) \quad (x_1 \notin \mathrm{dom}(\Gamma, x{:}T\,\widehat{\texttt{list}} \triangleright \Upsilon)) \quad (x_2 \notin \mathrm{dom}(\Gamma, x{:}T\,\widehat{\texttt{list}} \triangleright \Upsilon)) \\ \Gamma, x_2{:}T\,\texttt{list} \vdash \{x_1{:}T \triangleright \Upsilon \mid \exists x{:}T\,\texttt{list}.\varphi \wedge x_1 :: x_2 = x\}\ IS\ \{\Upsilon \mid \exists x{:}T\,\texttt{list}.\varphi \wedge x_2 = x\}\end{array}}{\Gamma \vdash \{x{:}T\,\texttt{list} \triangleright \Upsilon \mid \varphi\}\ \texttt{ITER}\, IS\ \{\Upsilon \mid \exists x{:}T\,\texttt{list}.\varphi \wedge x = []\}} \text{ (RT-ITER)}$$

$$\frac{\begin{array}{l}(x \notin \mathrm{dom}(\Gamma, \widehat{\Upsilon}) \cup \{y_1, y_1', y_2\}) \quad (y_1 \neq y_2) \quad y_1'{:}T_1, y_1{:}T_1 \vdash \varphi_1 : *\\ y_1'{:}T_1 \vdash \{y_1{:}T_1 \triangleright \ddagger \mid y_1' = y_1 \wedge \varphi_1\}\ IS\ \{y_2{:}T_2 \triangleright \ddagger \mid \varphi_2\}\end{array}}{\begin{array}{l}\Gamma \vdash \{\Upsilon \mid \varphi\}\ \texttt{LAMBDA}\, T_1\, T_2\, IS\ \{x{:}T_1 \to T_2 \triangleright \Upsilon \mid \varphi \wedge \forall y_1'{:}T_1, y_1{:}T_1, y_2{:}T_2.y_1' = y_1 \wedge \varphi_1 \wedge \\ \texttt{call}(x, y_1') = y_2 \implies \varphi_2\}\end{array}} \text{ (RT-LAMBDA)}$$

$$\frac{(x_3 \notin \mathrm{dom}(\Gamma, x_1{:}T_1 \triangleright \widehat{x_2{:}T_1 \to T_2} \triangleright \Upsilon))}{\Gamma \vdash \{x_1{:}T_1 \triangleright x_2{:}T_1 \to T_2 \triangleright \Upsilon \mid \varphi\}\ \texttt{EXEC}\ \{x_3{:}T_2 \triangleright \Upsilon \mid \exists x_1{:}T_1, x_2{:}T_1 \to T_2.\varphi \wedge \texttt{call}(x_2, x_1) = x_3\}} \text{ (RT-EXEC)}$$

$$\frac{(x_4 \notin \mathrm{dom}(\Gamma, x_1{:}T \triangleright x_2{:}\texttt{int} \triangleright \widehat{x_3{:}\texttt{address}} \triangleright \Upsilon))}{\begin{array}{l}\Gamma \vdash \{x_1{:}T \triangleright x_2{:}\texttt{int} \triangleright x_3{:}\texttt{address} \triangleright \Upsilon \mid \varphi\}\ \texttt{TRANSFER\_TOKENS}\, T\ \{x_4{:} \\ \texttt{operation} \triangleright \Upsilon \mid \exists x_1{:}T, x_2{:}\texttt{int}, x_3{:}\texttt{address}.\varphi \wedge x_4 = \texttt{Transfer}(x_1, x_2, x_3)\}\end{array}} \text{ (RT-TRANSFERTOKENS)}$$

$$\frac{\Gamma \vdash \Phi_1 <: \Phi_1' \qquad \Gamma \vdash \Phi_1'\ I\ \Phi_2' \qquad \Gamma \vdash \Phi_2' <: \Phi_2}{\Gamma \vdash \Phi_1\ I\ \Phi_2} \text{ (RT-SUB)}$$

Fig. 8 Typing rules (II)

(if it terminates). The post-condition in the conclusion expresses, by using `call`, that the function x has the property above. The extra variable y_1' in the type environment of the premise is an alias of y_1; being a variable declared in the type environment y_1' can appear in both φ_1 and φ_2[5] and can describe the relationship between the input and output of the function.

- (RT-EXEC) just adds to the post-condition $\texttt{call}(x_2, x_1) = x_3$, meaning the result of a call to the function x_2 with x_1 as an argument yields x_3. It may look simpler than expected; the crux here is that φ is expected to imply $\forall x_1 : T_1, x_3 : T_2.\varphi_1 \wedge \texttt{call}(x_2, x_1) = x_3 \implies \varphi_2$, where φ_1 and φ_2 represent the pre- and post-conditions, respectively, of function x_2. If x_1 satisfies φ_1, then we can derive that φ_2 holds.
- (RT-SUB) is the rule for subsumption to strengthening the pre-condition and weakening the post-condition.

[5] The scope of a variable in a refinement stack type is its predicate part and so y_1 cannot appear in the post-condition.

Stack Typing $\boxed{S : \bar{T}}$

$$\frac{}{\ddagger : \ddagger}\ (\text{ST-Bottom}) \qquad \frac{V : T \qquad S : \bar{T}}{V \triangleright S : T \triangleright \bar{T}}\ (\text{ST-Push})$$

Refinement Stack Typing $\boxed{\sigma : \Gamma \models S : \Phi}$

$$\frac{\sigma : \Gamma \models \varphi}{\sigma : \Gamma \models \ddagger : \{\ddagger \mid \varphi\}}\ (\text{Sem-Bottom}) \qquad \frac{\sigma[x \mapsto V] : \Gamma, x{:}T \models S : \{\Upsilon \mid \varphi\}}{\sigma : \Gamma \models V \triangleright S : \{x{:}T \triangleright \Upsilon \mid \varphi\}}\ (\text{Sem-Push})$$

Fig. 9 Simple and refinement stack typing

Properties

In this section, we show *soundness* of our type system. Informally, what we show is that, for a well-typed program, if we execute it under a stack which satisfies the pre-condition of the typing, then (if the evaluation halts) the resulting stack satisfies the post-condition of the typing. We only sketch proofs with important lemmas. The detailed proofs are found in Appendix A.

To state the soundness formally, we give additional definitions.

Definition 5 (Free variables). The set of free variables in φ is denoted by $\mathrm{fvars}(\varphi)$.

Definition 6 (Erasure) We define $\lfloor \Phi \rfloor$, which is the simple stack type obtained by erasing predicates from Φ, as follows.

$$\lfloor \{\ddagger \mid \varphi\} \rfloor = \ddagger \qquad \lfloor \{x : T \triangleright \Upsilon \mid \varphi\} \rfloor = T \triangleright \lfloor \{\Upsilon \mid \varphi\} \rfloor$$

Definition 7 (Stack typing). *Stack typing* $S : \bar{T}$ and *refinement stack typing* $\sigma : \Gamma \vDash S : \Phi$ are defined by the rules in Fig. 9.

Note that the definition of refinement stack typing follows the informal explanation of the refinement stack types in Section 'Refinement Type System'.

We start from soundness of simple type system as follows, that is, for a simply well-typed instruction sequence $\bar{T}_1 \vdash IS \Rightarrow \bar{T}_2$, if evaluation starts from correct stack S_1, that is $S_1 : \bar{T}_1$, and results in a stack S_2; then S_2 respects $\bar{T}_2$, that is $S_2 : \bar{T}_2$. This lemma is not only just a desirable property but also one we use for proving the soundness of the refinement type system in the case `EXEC`.

Lemma 8 (Soundness of the simple type system) *If* $\bar{T}_1 \vdash IS \Rightarrow \bar{T}_2$, $S_1 \vdash IS \Downarrow S_2$, *and* $S_1 : \bar{T}_1$, *then* $S_2 : \bar{T}_2$.

Proof Proved with a similar statement

If $\bar{T}_1 \vdash I \Rightarrow \bar{T}_2$, $S_1 \vdash I \Downarrow S_2$, and $S_1 : \bar{T}_1$, then $S_2 : \bar{T}_2$

for a single instruction by simultaneous induction on $\bar{T}_1 \vdash IS \Rightarrow \bar{T}_2$ and $\bar{T}_1 \vdash I \Rightarrow \bar{T}_2$. □

We state the main theorem as follows.

Theorem 9 (Soundness of the refinement type system) *If* $\Gamma \vdash \Phi_1\ IS\ \Phi_2$, $S_1 \vdash IS \Downarrow S_2$, *and* $\sigma : \Gamma \vDash S_1 : \Phi_1$, *then* $\sigma : \Gamma \vDash S_2 : \Phi_2$.

The proof is close to a proof of soundness of Hoare logic, with a few extra complications due to the presence of first-class functions. One of the key lemmas is the following one, which states that a value assignment can be represented by a logical formula or a stack element:

Lemma 10 *The following statements are equivalent*:

(1) $\sigma : \Gamma \vDash S : \{Y \mid \exists x : T.\varphi \wedge x = V\}$;
(2) $\sigma[x \mapsto V] : \Gamma, x : T \vDash S : \{Y \mid \varphi\}$; *and*
(3) $\sigma : \Gamma \vDash V \rhd S : \{x : T \rhd Y \mid \varphi\}$.

Then, we prove a few lemmas related to `LOOP` (Lemma 11), `ITER` (Lemma 12), predicate `call` (Lemmas 13 and 14), and subtyping (Lemma 15).

Lemma 11 *Suppose IS satisfies that* $S_1 \vdash IS \Downarrow S_2$ *and* $\sigma : \Gamma \vDash S_1 : \{Y \mid \exists x : \mathtt{int}.\varphi \wedge x \neq 0\}$ *imply* $\sigma : \Gamma \vDash S_2 : \{x : \mathtt{int} \rhd Y \mid \varphi\}$ *for any* S_1 *and* S_2. *If* $S_1 \vdash \mathtt{LOOP}\ IS \Downarrow S_2$ *and* $\sigma : \Gamma \vDash S_1 : \{x : \mathtt{int} \rhd Y \mid \varphi\}$, *then* $\sigma : \Gamma \vDash S_2 : \{Y \mid \exists x : \mathtt{int}.\varphi \wedge x = 0\}$.

Proof By induction on the derivation of $S_1 \vdash \mathtt{LOOP}\ IS \Downarrow S_2$. □

Lemma 12 *Suppose* $x_1 \notin \mathrm{fvars}(\varphi)$, $x_2 \notin \mathrm{fvars}(\varphi)$, *and that* $S_1' \vdash IS \Downarrow S_2'$ *and* $\sigma' : \Gamma, x_2 : T\ \mathtt{list} \vDash S_1' : \{x_1 : T \rhd Y \mid \exists x : T\ \mathtt{list}.\varphi \wedge x_1 :: x_2 = x\}$ *imply* $\sigma' : \Gamma, x_2 : T\ \mathtt{list} \vDash S_2' : \{Y \mid \exists x : T\ \mathtt{list}.\varphi \wedge x_2 = x\}$ *for any* S_1', S_2', *and* σ'. *If* $S_1 \vdash \mathtt{ITER}\ IS \Downarrow S_2$ *and* $\sigma : \Gamma \vDash S_1 : \{x : T\ \mathtt{list} \rhd Y \mid \varphi\}$, *then* $\sigma : \Gamma \vDash S_2 : \{Y \mid \exists x : T\ \mathtt{list}.\varphi \wedge x = []\}$.

Proof By induction on the derivation of $S_1 \vdash \mathtt{ITER}\ IS \Downarrow S_2$. □

Lemma 13 *If* $y_1 \neq y_2$, $y_1' : T_1, y_1 : T_1 \vdash \varphi_1 : *$, $y_1' : T_1, y_2 : T_2 \vdash \varphi_2 : *$, $\langle IS \rangle : T_1 \to T_2$, *and*

$$\begin{aligned}&\text{for any } V_1, V_2, \sigma, \text{ if } V_1 \rhd \ddagger \vdash IS \Downarrow V_2 \rhd \ddagger \text{ and}\\ &\qquad \sigma : y_1' : T_1 \vDash V_1 \rhd \ddagger : \{y_1 : T_1 \rhd \ddagger \mid y_1' = y_1 \wedge \varphi_1\}\\ &\qquad \text{then } \sigma : y_1' : T_1 \vDash V_2 \rhd \ddagger : \{y_2 : T_2 \rhd \ddagger \mid \varphi_2\},\end{aligned}$$

then $\Gamma \vDash \forall y_1' : T_1, y_1 : T_1, y_2 : T_2.y_1' = y_1 \wedge \varphi_1 \wedge \mathtt{call}(\langle IS \rangle, y_1') = y_2 \implies \varphi_2$ *for any* Γ.

Proof By the definition of the semantics of `call`. □

Lemma 14 *If* $V_1 \rhd \ddagger \vdash IS \Downarrow V_2 \rhd \ddagger$, $V_1 : T_1$, $V_2 : T_2$, *and* $\langle IS \rangle : T_1 \to T_2$, *then* $\Gamma \vDash \texttt{call}(\langle IS \rangle, V_1) = V_2$ *for any* Γ.

Proof By the definition of the semantics of `call`. □

Lemma 15 *If* $\Gamma \vdash \Phi_1 <: \Phi_2$ *and* $\sigma : \Gamma \vDash S : \Phi_1$, *then* $\sigma : \Gamma \vDash S : \Phi_2$.

Proof Straightforward from Definition 4. □

Proof of Theorem 9 It is proved together with a similar statement

If $\Gamma \vdash \Phi_1\ I\ \Phi_2, S_1 \vdash I \Downarrow S_2$, and $\sigma : \Gamma \vDash S_1 : \Phi_1$, then $\sigma : \Gamma \vDash S_2 : \Phi_2$.

for a single instruction by simultaneous induction on $\Gamma \vdash \Phi_1\ IS\ \Phi_2$ and $\Gamma \vdash \Phi_1\ I\ \Phi_2$ with case analysis on the last typing rule used. We show a few representative cases.

Case (RT-DIP): We have $I = \texttt{DIP}\, IS$ and $\Phi_1 = \{x : T \rhd Y \mid \varphi\}$ and $\Phi_2 = \{x : T \rhd Y' \mid \varphi'\}$ and $\Gamma, x : T \vdash \{Y \mid \varphi\}\ IS\ \{Y' \mid \varphi'\}$ for some IS, x, T, Y, Y', φ, and φ'. By (E-DIP), we have $S_1 = V \rhd S_1'$ and $S_2 = V \rhd S_2'$ and $S_1' \vdash IS \Downarrow S_2'$ for some V, S_1', and S_2'. By Lemma 10, we have $\sigma[x \mapsto V] : \Gamma, x : T \vDash S_1' : \{Y \mid \varphi\}$. By applying IH, we have $\sigma[x \mapsto V] : \Gamma, x : T \vDash S_2' : \{Y' \mid \varphi'\}$ from which $\sigma : \Gamma \vDash V \rhd S_2' : \{x : T \rhd Y' \mid \varphi'\}$ follows.

Case (RT-LOOP): We have $I = \texttt{LOOP}\, IS$ and $\Phi_1 = \{x : \texttt{int} \rhd Y \mid \varphi\}$ and $\Phi_2 = \{Y \mid \exists x : \texttt{int}.\varphi \wedge x = 0\}$ and $\Gamma \vdash \{Y \mid \exists x : \texttt{int}.\ \varphi \wedge x \neq 0\}\ IS\ \{x : \texttt{int} \rhd Y \mid \varphi\}$ and $S_1 = i \rhd S$ for some IS, x, Y, S, and φ. By IH, we have that, for any S_1', S_2', if $S_1' \vdash IS \Downarrow S_2'$ and $\sigma : \Gamma \vDash S_1' : \{Y \mid \exists x : \texttt{int}.\ \wedge \varphi_1 \wedge \texttt{call}(x, y_1') = y_2 \implies \varphi_2\}$ The goal easily follows from Lemma 11.

Case (RT-LAMBDA): We have $I = \texttt{LAMBDA}\, T_1\, T_2\, IS$ and $\Phi_1 = \{Y \mid \varphi\}$ and $\Phi_2 = \{x : T_1 \to T_2 \rhd Y \mid \varphi \wedge \forall y_1' : T_1, y_1 : T_1, y_2 : T_2.y_1' = y_1 \wedge \varphi_1 \wedge \texttt{call}(x, y_1') = y_2 \implies \varphi_2\}$ and $y_1' : T_1 \vdash \{y_1 : T_1 \rhd \ddagger \mid y_1' = y_1 \wedge \varphi_1\}\ IS\ \{y_2 : T_2 \rhd \ddagger \mid \varphi_2\}$ and $x \notin \mathrm{dom}(\Gamma, \widehat{Y}) \cup \{y_1, y_1', y_2\}$ and $y_1 \neq y_1'$ and $y_1' : T_1, y_1 : T_1 \vdash \varphi_1 : *$ and for some IS, x, y_1, y_1', y_2, T_1, T_2, Y, φ, φ_1, and φ_2. By (E-LAMBDA), we also have $S_2 = \langle IS \rangle \rhd S_1$. By IH, we have that, for any V_1, V_2, σ, if $V_1 \rhd \ddagger \vdash IS \Downarrow V_2 \rhd \ddagger$ and $\sigma : y_1' : T_1 \vDash V_1 \rhd \ddagger \mid y_1' = y_1 \wedge \varphi_1\}$, then $\sigma : y_1' : T_1 \vDash V_2 \rhd \ddagger : \{y_2 : T_2 \rhd \ddagger \mid \varphi_2\}$. By Lemma 13, $\Gamma, \widehat{Y} \vDash \forall y_1' : T_1, y_1 : T_1, y_2 : T_2.y_1' = y_1 \wedge \varphi_1 \wedge \texttt{call}(\langle IS \rangle, y_1') = y_2 \implies \varphi_2$. Then, it is easy to show

$\sigma : \Gamma \vDash S_1 : \{Y \mid \varphi \wedge \forall y_1' : T_1, y_1 : T_1, y_2 : T_2.y_1' = y_1 \wedge \varphi_1 \wedge \texttt{call}(\langle IS \rangle, y_1') = y_2 \implies \varphi_2\}$. Then, we have

$$\sigma : \Gamma \vDash S_1 : \{Y \mid \exists x : T_1 \to T_2.(\varphi \wedge \forall y_1' : T_1, y_1 : T_1, y_2 : T_2.y_1' = y_1 \wedge \varphi_1 \wedge \texttt{call}(x, y_1') = y_2 \implies \varphi_2) \wedge x = \langle IS \rangle\}.$$

Therefore, by Lemma 10, we have

$$\sigma : \Gamma \vDash \langle IS \rangle \rhd S_1 : \{x : T_1 \to T_2 \rhd Y \mid \varphi \wedge \forall y_1' : T_1, y_1 : T_1, y_2 : T_2.y_1' = y_1 \wedge \varphi_1 \wedge \texttt{call}(x, y_1') = y_2 \implies \varphi_2\}$$

as required.

Case (RT-Exec): We have $I = \texttt{EXEC}$ and, for some x_1, x_2, x_3, T_1, T_2, Y, and φ, $\Phi_1 = \{x_1 : T_1 \rhd x_2 : T_1 \to T_2 \rhd Y \mid \varphi\}$ and $\Phi_2 = \{x_3 : T_2 \rhd Y \mid \exists x_1 : T_1, \widehat{x_2 : T_1} \to T_2.\varphi \wedge \texttt{call}(x_2, x_1) = x_3\}$ and $x_3 \notin \text{dom}(\Gamma, x_1 : T_1 \rhd \widehat{x_2 : T_1} \to T_2 \rhd Y)$. By (E-Exec), we have $S_1 = V_1 \rhd \langle IS \rangle \rhd S$ and $S_2 = V_2 \rhd S$ and $V_1 \rhd \ddagger \vdash IS \Downarrow V_2 \rhd \ddagger$ for some V_1, V_2, IS, and S. By the assumption $\sigma : \Gamma \vDash S_1 : \Phi_1$, we have $\sigma : \Gamma \vDash S : \{Y \mid \exists x_2 : T_1 \to T_2.(\exists x_1 : T_1.\varphi \wedge x_1 = V_1) \wedge x_2 = \langle IS \rangle\}$. By Lemma 14, we have $\Gamma, \widehat{Y} \vDash \texttt{call}(\langle IS \rangle, V_1) = V_2$. Therefore, we have $\sigma : \Gamma \vDash S : \{Y \mid (\exists x_2 : T_1 \to T_2.(\exists x_1 : T_1.\varphi \wedge x_1 = V_1) \wedge x_2 = \langle IS \rangle) \wedge \texttt{call}(\langle IS \rangle, V_1) = V_2\}$. Finally, we have $\sigma : \Gamma \vDash S : \{Y \mid \exists x_3 : T_2.(\exists x_1 : T_1, x_2 : T_1 \to T_2.\varphi \wedge \texttt{call}(x_2, x_1) = x_3) \wedge x_3 = V_2\}$ and, thus, $\sigma : \Gamma \vDash V_2 \rhd S : \{x_3 : T_2 \rhd Y \mid (\exists x_1 : T_1, x_2 : T_1 \to T_2.\varphi \wedge \texttt{call}(x_2, x_1) = x_3)\}$ as required.

Case (RT-Sub): We have $\Gamma \vdash \Phi_1 <: \Phi_1'$ and $\Gamma \vdash \Phi_2' <: \Phi_2$ and $\Gamma \vdash \Phi_1' \; I \; \Phi_2'$ for some Φ_1' and Φ_2'. By Lemma 15, we have $\sigma : \Gamma \vDash S_1 : \Phi_1'$. By IH, we have $\sigma : \Gamma \vDash S_2 : \Phi_2'$. Then, the goal follows from Lemma 15. □

Extension with Exceptions

The type system implemented in Helmholtz is extended to handle instruction `FAILWITH`, which immediately aborts the execution, discarding all the stack elements but the top element. The type judgment form is extended to

$$\Gamma \vdash \Phi_1 \; IS \; \Phi_2 \; \& \; \Phi_3,$$

which means that, if IS is executed under a stack satisfying Φ_1, then the resulting stack satisfies Φ_2 (if normally terminates), or Φ_3 (if aborted by `FAILWITH`). The typing rule for instruction `FAILWITH`, which raises an exception with the value at the stack top, is given as follows:

$$\Gamma \vdash \{x : T \rhd Y \mid \varphi\} \; \texttt{FAILWITH} \; \{Y \mid \bot\} \; \& \; \{\texttt{err} \mid \exists x : T, \widehat{Y}.\varphi \wedge x = \texttt{err}\}.$$

$$\frac{(x \notin \mathrm{dom}(\Gamma, \widehat{\Upsilon}) \cup \{y_1, y_1', y_2\}) \quad (y_1 \neq y_2) \quad y_1' : T_1, y_1 : T_1 \vdash \varphi_1 : * \quad y_1' : T_1 \vdash \{y_1 : T_1 \triangleright \ddagger \mid y_1' = y_1 \wedge \varphi_1\}\ IS\ \{y_2 : T_2 \triangleright \ddagger \mid \varphi_2\}\ \&\ \{\mathtt{err} \mid \varphi_3\}}{\Gamma \vdash \{\Upsilon \mid \varphi\}\ \mathtt{LAMBDA}\ T_1\ T_2\ IS\ \{x : T_1 \to T_2 \triangleright \Upsilon \mid \varphi \wedge \forall y_1' : T_1, y_1 : T_1, y_2 : T_2 . y_1' = y_1 \wedge \varphi_1 \implies (\mathtt{call}(x, y_1') = y_2 \implies \varphi_2) \wedge (\mathtt{call\_err}(x, y_1') = \mathtt{err} \implies \varphi_3)\}\ \&\ \{\mathtt{err} \mid \bot\}}$$

$$\frac{(x_3 \notin \mathrm{dom}(\Gamma, x_1 : T_1 \triangleright \widehat{x_2 : T_1 \to T_2 \triangleright \Upsilon}))}{\Gamma \vdash \{x_1 : T_1 \triangleright x_2 : T_1 \to T_2 \triangleright \Upsilon \mid \varphi\}\ \mathtt{EXEC}\ \{x_3 : T_2 \triangleright \Upsilon \mid \exists x_1 : T_1, x_2 : T_1 \to T_2 . \varphi \wedge \mathtt{call}(x_2, x_1) = x_3\}\ \&\ \{\mathtt{err} \mid \exists x_1 : T_1, x_2 : T_1 \to T_2, \widehat{\Upsilon} . \varphi \wedge \mathtt{call\_err}(x_2, x_1) = \mathtt{err}\}}$$

Fig. 10 Modified typing rules for LAMBDA and EXEC

The rule expresses that, if FAILWITH is executed under a non-empty stack that satisfies φ, then the program point just after the instruction is not reachable (hence, $\{\Upsilon \mid \bot\}$). The refinement $\exists x : T, \widehat{Y} . \varphi \wedge x = \mathtt{err}$ for the exception case states that φ in the pre-condition with the top element x is equal to the raised value err; since x is not in the scope in the exception refinement, x is bound by an existential quantifier. Most of the other typing rules can be extended with the " &" part easily.

For (RT-LAMBDA) and (RT-EXEC), we first extend the the assertion language with a new predicate $\mathtt{call\_err}(t_1, t_2) = t_3$ meaning the call of t_1 with t_2 aborts with the value t_3. (The semantics of call is unchanged.) Using the new predicate, (RT-LAMBDA) and (RT-EXEC) are modified as in Fig. 10.

Tool Implementation

In this section, we present HELMHOLTZ, the verification tool based on the refinement type system. We first discuss how Michelson code can be annotated. Then, we give an overview of the verification algorithm, which reduces the verification problem to SMT solving, and discuss how Michelson-specific features are encoded. Finally, we show a case study of contract verification and present verification experiments.

Annotations

HELMHOLTZ supports several forms of annotations (surrounded by << and >> in the source code), other than ContractAnnot explained in Section 'Overview of HELMHOLTZ and Michelson'. As we have already seen, the syntax of refinement stack types used in the implementation is slightly different from the formal definition: we use a colon-separated list of ML-like patterns to bind stack values and ML-like expressions to describe the predicate, which have to be quantifier free (mainly because state-of-the-art SMT solvers do not handle quantifiers very well). We explain several constructs for an annotation in the following.

Assert Φ and Assume Φ can appear before or after an instruction. The former asserts that the stack at the annotated program location satisfies the type Φ; the

```
parameter unit ;
storage int;
<< ContractAnnot { _ | True } -> { _ | True }
                 & { _ | False } >>
code { DROP;
       << LambdaAnnot { p | p = (3, 1) } -> { x | x = 4 }
                      & { _ | False }
          (a:int, b:int) >>
       LAMBDA (pair int int) int
              { << Assume { p | p = (a, b) } >>
                UNPAIR; ADD
                << Assert { p | p = a + b } >>
              };
       PUSH int 1; PUSH int 3; PAIR; EXEC;
       << Assert { x | x = 4 } >>
       NIL operation; PAIR
     }
```

```
parameter (list int);
storage int;
<< Measure len : list int -> int
     where [] = 0 | h :: t = (1 + len t) >>
<< ContractAnnot
   { (p, _) | True } -> { (_, ret) | len p = ret }
   & { _ | False } >>
code { CAR; PUSH int 0; SWAP;
       << LoopInv { l : n | len l + n = len p } >>
       ITER { DROP; PUSH int 1; ADD };
       NIL operation;
       PAIR
     }
```

Fig. 11 `lambda.tz`, which uses higher-order functions, and `length.tz`, which uses a measure function in the contract annotation

assertion is verified by Helmholtz . If there is an annotation `Assume` Φ, Helmholtz assumes that the stack satisfies the type Φ at the annotated program location. A user can give a hint to Helmholtz by using `Assume` Φ. The user has to make sure that it is correct; if an `Assume` annotation is incorrect, the verification result may also be incorrect.

An annotation `LoopInv` Φ may appear before a loop instruction (e.g., `LOOP` and `ITER`). It asserts that Φ is a loop invariant of the loop instruction. In the current implementation, annotating a loop invariant using `LoopInv` Φ is mandatory for a loop instruction. Helmholtz checks that Φ is indeed a loop invariant and uses it to verify the rest of the program.

In the current implementation, a `LAMBDA` instruction, which pushes a function on the top of the stack, must be accompanied by a `LambdaAnnot` annotation. `LambdaAnnot` comes with a specification of the pushed function written in the same way as `ContractAnnot`. Concretely, the specification of the form $\Phi_{\text{pre}} \rightarrow \Phi_{\text{post}}$ & $\Phi_{\text{abpost}}(x_1 : T_1, \ldots, x_n : T_n)$ specifies the precondition Φ_{pre}, the (normal) postcondition Φ_{post}, and the (abnormal) postcondition Φ_{abpost} as refinement

$$\frac{}{[] \triangleright S \vdash \texttt{MAP}\, IS \Downarrow [] \triangleright S}\ (\text{E-MapNil}) \qquad \frac{V_1 \triangleright S \vdash IS \Downarrow V_1' \triangleright S' \qquad V_2 \triangleright S' \vdash \texttt{MAP}\, IS \Downarrow V_2' \triangleright S''}{V_1 :: V_2 \triangleright S \vdash \texttt{MAP}\, IS \Downarrow V_1' :: V_2' \triangleright S''}\ (\text{E-MapCons})$$

$$\frac{\Gamma, x_2{:}T\,\texttt{list}, y_2{:}T\,\texttt{list} \vdash \{x_1{:}T \triangleright \Upsilon \mid \exists z{:}T\,\texttt{list}, z'{:}T\,\texttt{list}.\varphi \wedge z = x_1 :: x_2 \wedge z' = y_2\}\ IS\ \{y_1{:}T \triangleright \Upsilon \mid \exists z{:}T\,\texttt{list}, z'{:}T\,\texttt{list}.\varphi \wedge z = x_2 \wedge z' = y_2 @ [y_1]\}}{\Gamma \vdash \{z{:}T\,\texttt{list} \triangleright \Upsilon \mid \exists z'{:}T\,\texttt{list}.\varphi \wedge z' = []\}\ \texttt{MAP}\, IS\ \{z'{:}T\,\texttt{list} \triangleright \Upsilon \mid \exists z{:}T\,\texttt{list}.\varphi \wedge z = []\}}\ (\text{RT-Map})$$

Fig. 12 Operational semantics and typing rules for MAP

stack types. The binding $(x_1 : T_1, \ldots, x_n : T_n)$ introduces the ghost variables that can be used in the annotations in the body of the annotated LAMBDA instruction;[6]; one can omit if it is empty.

The first contract in Fig. 11, which pushes a function that takes a pair of integers and returns the sum of them, exemplifies LambdaAnnot. The annotated type of the function (Line 6) expresses that it returns 4 if it is fed with a pair (3, 1). The ghost variables *a* and *b* are used in the annotations Assume (Line 10) and Assert (Line 12) in the body to denote the first and the second arguments of the pair passed to this function.

To describe a property for recursive data structures, Helmholtz supports *measure functions* introduced by Kawaguchi et al. [11] and also supported in Liquid Haskell [24]. Measure functions are one of the potent techniques to handle such properties without universal quantifiers. (Recall that we should avoid quantifiers as much as possible.) A measure function is a (recursive) function over a recursive data structure that can be used in assertions. The annotation Measure $x : T_1 \to T_2$ where $p_1 = e_1 \mid \ldots \mid p_n = e_n$ defines a measure function x over the type T_1. The measure function x takes a value of type T_1, destructs it by the pattern matching, and returns a value of type T_2. Metavariables p and e represent ML-like patterns and expressions. The second contract in Fig. 11, which computes the length of the list passed as a parameter, exemplifies the usage of the Measure annotation. This contract defines a measure function len that takes a list of integers and returns its length; it is used in ContractAnnot and LoopInv.

Overview of the Verification Algorithm

Helmholtz takes an annotated Michelson program and conducts typechecking based on the refinement type system in Section 'Refinement Type System'. All the instructions except for MAP can be dealt with by a straightforward extension of the present typing rules. One exception is MAP, which is a Michelson instruction for applying a function to each element in the list or the associative array at the top of the stack. Figure 12 shows the operational semantics and typing rules for MAP on lists, where @ denotes the list concatenation operator. They are similar to those for ITER, but the fact that MAP manipulates and pushes back each

[6] ContractAnnot also allows declarations of ghost variable used in the code section.

element of the list[7] (cf. V_1 to V_1' and V_2 to V_2' in (E-MapCons)) makes differences. In the typing rule (RT-Map), z' stands for the manipulated list. So, it is assumed that $z' = []$ at the beginning of the execution of `MAP`. One non-trivial issue is that, unlike (RT-Loop) or (RT-Iter), the precondition itself is *not* a loop invariant because, in the middle of the manipulation, $z' = []$ does not hold. Although we do not show a proof, the typing rule (RT-Map) is sound. Intuitively, it is because `MAP` can be simulated by `ITER`, `DIP`, and other instructions, and the rule (RT-Map) is obtained from the typing derivation of the simulating instruction sequence.

The typechecking procedure (1) computes the verification conditions (VCs) for the program to be well typed and (2) discharges it using an SMT solver. The latter is standard: We decide the validity of the generated VCs using an SMT solver (Z3 in the current implementation.) We explain the VC-generation step in detail.

For an annotated contract, Helmholtz conducts forward reasoning starting from the precondition and generates VCs if necessary. During the forward reasoning, Helmholtz keeps track of the Γ-and-Y part of the type judgment.

The typing rules are designed so that they enable the forward reasoning if a program is simply typed. For example, consider the rule (RT-Add) in Fig. 7. This rule can be read as a rule to compute a postcondition $\exists x_1 : \mathtt{int}, x_2 : \mathtt{int}.\varphi \wedge x_1 + x_2 = x_3$ from a precondition φ if the first two elements in Y are x_1 and x_2. The other rules can also be read as postcondition-generation rules in the same way.

There are three places where Helmholtz generates a verification condition.

- At the end of the program: Helmholtz generates a condition that ensures that the computed postcondition of the entire program implies the postcondition annotated to the program.
- Before and after instruction `LAMBDA`: Helmholtz generates conditions that ensure that the pre- and post-conditions of the instruction `LAMBDA` is as annotated in `LambdaAnnot`.
- At a loop instruction: Helmholtz generates verification conditions that ensure the condition annotated by `LoopInv` is indeed a loop invariant of this instruction.

A VC generated by Helmholtz at these places is of the form $\forall \mathbf{x} : \mathbf{T}.\varphi_1 \Longrightarrow \varphi_2$, where $\mathbf{x} : \mathbf{T}$ is a sequence of bindings.

To discharge each VC, as many verification condition discharging procedures do, Helmholtz checks whether its negation, $\exists \mathbf{x} : \mathbf{T}.\varphi_1 \wedge \neg\varphi_2$, is satisfiable; if it is unsatisfiable, then the original VC is successfully discharged. We remark that our type system is designed so that φ_1 and φ_2 are quantifier free for a program that does not use `LAMBDA` and `EXEC`. Indeed, φ_2 comes only from the annotations,

[7] It can also manipulate the rest of the stack like `ITER`, which is slightly different from the behavior of a typical *map* in a functional programming language.

```
parameter (pair signature string);
storage (pair address key);
<< ContractAnnot
   { ((sign, data), (addr, pubkey)) |
       match contract_opt addr with
       | Some (Contract<string> _) -> True | _ -> False } ->
   { (ops, new_store) | (addr, pubkey) = new_store &&
              sig pubkey sign (pack data) &&
              match contract_opt addr with
              | Some c -> ops = [ TransferTokens data 1 c ]
              | None -> False }
   & { _ | not (sig pubkey sign (pack data)) } >>
code  { DUP; DUP; DUP;
        DIP { CAR; UNPAIR; DIP { PACK } }; CDDR;
        CHECK_SIGNATURE; ASSERT;

        UNPAIR; CDR; SWAP; CAR;
        CONTRACT string; ASSERT_SOME; SWAP;
        PUSH mutez 1; SWAP;
        TRANSFER_TOKENS;

        NIL operation; SWAP;
        CONS; DIP { CDR };
        PAIR
      }
```

Fig. 13 `checksig.tz`, which involves signature verification

which are quantifier-free. φ_1 comes from the postcondition-computation procedure, which is of the form $\exists \mathbf{x}' : \mathbf{T}'.\varphi_1'$ for quantifier-free φ_1' for the instructions other than `LAMBDA` and `EXEC`; the formula $\exists \mathbf{x} : \mathbf{T}.\varphi_1 \wedge \neg\varphi_2$ is equivalent to $\exists \mathbf{x} : \mathbf{T}, \mathbf{x}' : \mathbf{T}'.\varphi_1' \wedge \neg\varphi_2$.

Encoding Michelson-Specific Features

Since our assertion language includes several specific features that originates from Michelson and our assertion language, Helmholtz needs to encode them in discharging VCs so that Z3 can handle them. We explain how this encoding is conducted.

Michelson-Specific Functions and Predicates

We encode several Michelson-specific functions using uninterpreted functions. For example, Helmholtz assumes the following typing rule for instruction `SHA256`, which converts the top element to its SHA256 hash.

$$\Gamma \vdash \{x : \texttt{bytes} \rhd Y \mid \phi\}\ \texttt{SHA256}\ \{y : \texttt{bytes} \rhd Y \mid \exists x.\phi \wedge y = \text{sha256}(x)\}.$$

In the post condition, we use an uninterpreted function sha256 to express the SHA256 hash of x. In Z3, this uninterpreted function is declared as follows.

```
1  (declare-fun sha256 (String) String)
2  (assert (forall ((x String)) (= (str.len (sha256 x)) 32)))
```

The first line declares the signature of sha256. The second line is the axiom for sha256 that the length of a hash is always 32.

Notice that, Helmholtz cannot prove that a calculated hash is equal to a specific constant since the represented uninterpreted function has under-specified axioms. For instance, Helmholtz cannot prove:

$$\Gamma \vdash \{x : \texttt{bytes} \mid x = 0\}\ \texttt{SHA256}\ \{y : \texttt{bytes} \rhd Y \mid \exists x.\phi \wedge y = \text{“6e340b9cffb37a989ca544e6bb780a2c78901d3fb33738768511a30617afa01d”}\}.$$

Nevertheless, practical contracts still can be verified with this light-weight axiomatization. For example, in Fig. 13, we use the `sig` uninterpreted function, which is given no specific axiom. Despite of the fact, the contract can be verified because the typing rule for `CHECK_SIGNATURE` in Line 15 ensures the stack top (Boolean value) after the instruction is equal to `sig pubkey sign (pack data)` in Line 8, and then, `ASSERT` just after the instruction ensures the top value is `True` in normal termination. That is why Helmholtz can verify the contract without the behavior of the `sig` function itself.

Implementation of Measure Functions

A measure function is encoded as an uninterpreted function accompanied with axioms that specify the behavior of the function, which is defined by the `Measure` annotation. Let us consider, for example, about the following measure function for lists.

```
1  << Measure f : list T -> T' where [] = e1 | h :: t = e2 >>
```

Theoretically, one could insert the following declarations and assertions when it generates an input to Z3 to encode this definition:

```
1  (declare-fun f ((List T)) T')
2  (assert (= (f []) e1))
3  (assert (forall ((h T) (t (List T)))
4     (= (f (cons h t)) e2)))
```

However, Z3 tends to timeout if we naively insert the above axioms to Z3 input which contains a universal quantifier in the encoded definition.

To address this problem, Helmholtz rewrites VCs so that heuristically instantiated conditions on a measure function are available where necessary. Consider the above definition of `f` as an example. Suppose Helmholtz obtains a VC of the form

$\exists \mathbf{x} : \mathbf{T}.\varphi_1 \wedge \neg\varphi_2$ mentioned in Section 'Overview of the Verification Algorithm' and φ_1 and φ_2 contains `(cons` $e_{h,i}$ $e_{t,i}$`)` for $i \in \{1, \ldots, N\}$. Then, Helmholtz constructs a formula $\varphi_{meas} := \bigwedge_{i\in\{1,\ldots,N\}} (= (\texttt{f(cons}\ e_{h,i}\ e_{t,i}\texttt{))e2})$ and rewrites the VC to $\exists \mathbf{x} : \mathbf{T}.\varphi_{meas} \wedge \varphi_1 \wedge \neg\varphi_2$.

We remark that, in LiquidHaskell, measure functions are treated as a part of the type system [11]: the asserted axioms are systematically (instantiated and) embedded into the typing rules. In Helmholtz , measure functions are treated as an ingredient that is orthogonal to the type system; the type system is oblivious of measure functions until its definition is inserted to Z3 input.

Overloaded Functions

Due to the polymorphically typed instructions in Michelson, our assertion language incorporates polymorphic uninterpreted functions. For example, Michelson has an instruction `PACK`, which pops a value (of any type) from the stack, serializes it to a binary representation, and pushes the serialized value. Helmholtz typechecks this instruction based on the following rule.

$$\Gamma \vdash \{x : T \rhd Y \mid \varphi\}\ \texttt{PACK}\ \{y : \texttt{bytes} \rhd Y \mid \exists x : T.\varphi \wedge y = \mathrm{pack}(x)\}$$

The term $\mathrm{pack}(x)$ in the postcondition represents a serialized value created from x. Since x may be of any simple type T, pack must be polymorphic.

Having a polymorphic uninterpreted function in assertions is tricky because Z3 does not support a polymorphic value. Helmholtz encodes polymorphic uninterpreted functions to a monomorphic function whose name is generated by mangling the name of its instantiated parameter type. For example, the above $\mathrm{pack}(x)$ is encoded as a Z3 function `pack!int`(x) whose type is `int` $\rightarrow$ `bytes`. Although there are infinitely many types, the number of the encoded functions is finite since only finitely many types appear in a single contract.

Michelson-Specific Types

In encoding a VC as Z3 constraints, Helmholtz maps types in Michelson into sorts in Z3, e.g., the Michelson type `nat` for nonnegative integers to the Z3 sort `Int`. A naive mapping from Michelson types to Z3 sorts is problematic; for example, $\forall x : \texttt{nat}.x \geq 0$ in Helmholtz is valid, but a naively encoded formula `(forall ((x Int)) (>= x 0))` is invalid in Z3. This naive encoding ignores that a value of sort `nat` is nonnegative.

To address the problem, we adapt the method encoding a many-sorted logic formula into a single-sorted logic formula [5]. Concretely, we define a *sort predicate* $P_T(x)$ for each sort T, which characterizes the property of the sort T. For example, $P_{\texttt{nat}}(x) := x \geq 0$. We also define sort predicates for compound data types.

Using the sort predicates, we can encode a VC into a Z3 constraint as follows: $\forall x : T.\phi$ is encoded into $\forall x : [\![T]\!].P_T(x) \Rightarrow \phi$ and $\exists x : T.\phi$ is encoded into $\exists x : [\![T]\!].P_T(x) \wedge \phi$, where $[\![T]\!]$ denotes the target sort of T (e.g., $[\![\texttt{nat}]\!] = \texttt{Int}$).

Table 1 Benchmark result

Filename	#instr.	Time (ms)	Filename	#instr.	Time (ms)
`boomerang.tz`	17	435	`checksig_unverified.tz`	30	462
`deposit.tz`	24	451	`vote_for_delegate.tz`	78	608
`manager.tz`	24	449	`xcat.tz`	52	513
`vote.tz`	24	450	`reservoir.tz`	45	482
`tzip.tz`	537	15578	`triangular_num.tz`	16	517
`checksig.tz`	32	468			

Furthermore, we also add axioms about co-domain of uninterpreted functions as $\forall \mathbf{x_i} : \mathbf{T_i}.P_T(f(\mathbf{x_i}))$ for the function f of $\mathbf{T_i} \to T$.

Case Study: Contract with Signature Verification

Figure 13 presents the code of the contract `checksig.tz`, which verifies that a sender indeed signed certain data using her private key. This contract uses instruction `CHECK_SIGNATURE`, which is supposed to be executed under a stack of the form `key` ▷ `sig` ▷ `bytes` ▷ `tl`, where `key` is a public key, `sig` is a signature, and `bytes` is some data. `CHECK_SIGNATURE` pops these three values from the stack and pushes `True` if `sig` is the valid signature for `bytes` with the private key corresponding to `key`. The instruction `ASSERT` after `CHECK_SIGNATURE` checks if the signature checking has succeeded; it aborts the execution of the contract if the stack top is `False`; otherwise, it pops the stack top (`True`) and proceeds the next instruction.

The intended behavior of `checksig.tz` is as follows. It stores a pair of an address `addr`, which is the address of a contract that takes a `string` parameter, and a public key `pubkey` in its storage. It takes a pair `(sign,data)` of type `(pair signature string)` as a parameter; here, `signature` is the primitive Michelson type for signatures. This contract terminates without exception if `sign` is created from the serialized (packed) representation of `data` and signed by the private key corresponding to `pubkey`. In a normal termination, this contract transfers 1 `mutez` to the contract with address `addr`. If this signature verification fails, then an exception is raised.

This behavior is expressed as a specification in the `ContractAnnot` annotation in `checksig.tz` as follows.

– The refinement of its pre-condition part expresses that the address stored in the first element `addr` of the storage is an address of a contract that takes a value of type `string` as a parameter. This is expressed by the pattern-matching on `contract_opt addr`, which checks if there is an intended parameter type of contract stored at the address `addr` and returns the contract (wrapped by `Some`) if there is. The intended parameter type is given by the pattern expression `Contract < string > _`, which matches a contract that takes a `string`.

– The refinement of the post-condition forces the following three conditions: (1) the store is not updated by this contract (`((addr, pubkey) = new_store)`); (2) `sign` is the signature created from the packed bytes `pack data` of the string in the second element of the parameter and signed by the private key corresponding to the second element `pubkey` of the store (`sig pubkey sign (pack data)`); and (3) the operations `ops` returned by this contract is `[Transfer data 1 c]`, which represents an operation of transferring 1 `mutez` to the contract `c`, which is bound to `Contract addr`, with the parameter `data`. The predicate `sig` and the function `pack` are primitives of the assertion language of Helmholtz .
– The refinement in the exception part expresses that if an exception is raised, then the signature verification should have failed (`not (sig pubkey sign (pack data))`).

Helmholtz successfully verifies `checksig.tz` without any additional annotation in the `code` section. If we change the instruction `ASSERT` in Line 15 to `DROP` to let the contract drop the result of the signature verification (hence, an exception is not raised even if the signature verification fails), the verification fails as intended.

Experiments

We applied Helmholtz to various contracts; Table 1 is an excerpt of the result, in which we show (1) the number of the instructions in each contract (column #instr.) and (2) time (ms) spent to verify each contract. The experiments are conducted on MacOS Big Sur 11.4 with Quad-Core Intel Core i7 (2.3 GHz), 32 GB RAM. We used Z3 version 4.8.10. The contracts `boomerang.tz`, `deposit.tz`, `manager.tz`, `vote.tz`, and `reservoir.tz` are taken from the benchmark of Mi-cho-coq [3]. `checksig.tz`, discussed above, is derived from `weather_insurance.tz` of the official Tezos test suite.[8] `vote_for_delegate.tz` and `xcat.tz` are taken from the official test suite; `xcat.tz` is simplified from the original. `tzip.tz` is taken from Tezos Improvement proposals.[9] `triangular_num.tz` is a simple test case that we made as an example of using `LOOP`. The source code of these contracts can be found at the Web interface of Helmholtz . Each contract is supposed to work as follows.

– `boomerang.tz`: Transfers the received amount of money to the source account.
– `deposit.tz`: Transfers money to the sender if the address of the sender is identical to that is stored in the storage.
– `manager.tz`: Calls the passed function if the address of the caller matches the address stored in the storage.

[8] https://gitlab.com/tezos/tezos/-/tree/ee2f75bb941522acbcf6d5065a9f3b2/tests_python/contracts/mini_scenarios.
[9] https://gitlab.com/tezos/tzip/-/blob/b73c7cd5df8e045bbf7ad9ac20a45fb3cb862c87/proposals/tzip-7/tzip-7.md.

- `vote.tz`: Accepts a vote to a candidate if the voter transfers enough voting fee, and stores the tally.
- `tzip.tz`: One of the components implementing Tezos smart contracts API. We verify one entrypoint of the contract.
- `checksig.tz`: The one explained in Section 'Case Study: Contract with Signature Verification'.
- `vote_for_delegate.tz`: Delegates one's ballot in voting by stakeholders, which is one of the fundamental features of Tezos, to another using a primitive operation of Tezos.
- `xcat.tz`: Transfers all stored money to one of the two accounts specified beforehand if called with the correct password. The account that gets money is decided based on whether the contract is called before or after a deadline.
- `reservoir.tz`: Sends a certain amount of money to either a contract or another depending on whether the contract is executed before or after the deadline.
- `triangular_num.tz`: Calculates the sum from 1 to n, which is the passed parameter.

In the experiments, we verified that each contract indeed works according to the intention explained above. `triangular_num.tz` was the only contract that required a manual annotation for verification in the `code` section; we needed to specify a loop invariant in this contract.

Although the numbers of instructions in these contracts are not large, they capture essential features of smart contracts; every contract except `triangular_num.tz` executes transactions; `deposit.tz` and `manager.tz` check the identity of the caller; and `checksig.tz` conducts signature verification. The time spent on verification is small.

Related Work

There are several publications on the formalization of programming languages for writing smart contracts. Hirai [9] formalizes EVM, a low-level smart contract language of Ethereum and its implementation, using Lem [15], a language to specify semantic definitions; definitions written in Lem can be compiled into definitions in Coq, HOL4, and Isabelle/HOL. Based on the generated definition, he verifies several properties of Ethereum smart contracts using Isabelle/HOL. Bernardo et al. [3] implemented Mi-Cho-Coq, a formalization of the semantics of Michelson using the Coq proof assistant. They also verified several Michelson contracts. Compared to their approach, we aim to develop an automated verification tool for smart contracts. Park et al. [16] developed a formal verification tool for EVM by using the K-framework [18], which can be used to derive a symbolic model checker from a formally specified language semantics (in this case, formalized EVM semantics [8]), and successfully applied the derived model checker to a few EVM contracts. It would be interesting to formalize the semantics of Michelson in the K-framework to compare Helmholtz with the derived model checker.

The DAO attack [19], mentioned in Section 'Introduction', is one of the notorious attacks on a smart contract. It exploits a vulnerability of a smart contract that is related to a callback. Grossman et al. [7] proposed a type-based technique to verify that execution of a smart contract that may contain callbacks is equivalent to another execution without any callback. This property, called *effectively callback freedom*, can be seen as one of the criteria for execution of a smart contract not to be vulnerable to the DAO-like attack. Their type system focuses on verifying the ECF property of the *execution* of a smart contract, whereas ours concerns the verification of generic functional properties of a smart contract.

Benton proposes a program logic for a minimal stack-based programming language [2]. His program logic can give an assertion to a stack as our stack refinement types do. However, his language does not support first-class functions nor instructions for dealing with smart contracts (e.g., signature verification).

Our type system is an extension of the Michelson type system with refinement types, which have been successfully applied to various programming languages [1, 11, 12, 17, 21, 23–28]. DTAL [27] is a notable example of an application of refinement types to an assembly language, a low-level language like Michelson. A DTAL program defines a computation using registers; we are not aware of refinement types for stack-based languages like Michelson.

We notice the resemblance between our type system and a program logic for PCF proposed by Honda and Yoshida [10], although the targets of verification are different.

Their logic supports a judgment of the form $A \vdash e : {}_{u}B$, where e is a PCF program, A is a pre-condition assertion, B is a post-condition assertion, and u represents the value that e evaluates to and can be used in B, which resembles our type judgment in the formalization in Section 'Refinement Type System for Mini-Michelson'. Their assertion language also incorporates a term expression $f \bullet x$, which expresses the value resulting from the application of f to x; this expression resembles the formula $\texttt{call}(t_1, t_2) = t_3$ used in a refinement predicate. We have not noticed an automated verifier implemented based on their logic. Further comparison is interesting future work.

Conclusion

We described our automated verification tool Helmholtz for the smart contract language Michelson based on the refinement type system for Mini-Michelson . Helmholtz verifies whether a Michelson program follows a specification given in the form of a refinement type. We also demonstrated that Helmholtz successfully verifies various practical Michelson contracts.

Currently, Helmholtz supports approximately 80% of the whole instructions of the Michelson language. The definition of a measure function is limited in the sense that, for example, it can define only a function with one argument. We are currently extending Helmholtz so that it can deal with more programs.

Helmholtz currently verifies the behavior of a single contract, although a blockchain application often consists of multiple contracts in which contract calls

are chained. To verify such an application as a whole, we plan to extend Helmholtz so that it can verify an inter-contract behavior compositionally by combining the verification results of each contract.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s00354-022-00167-1.

Acknowledgements This work has been partially supported by a research grant from Tezos Foundations. Igarashi's first encounter with formal proofs was when he visited Prof. Hagiya's group as an intern and played Boomborg PC.

Funding This work has been partially supported by a research grant from Tezos Foundations.

Availability of data and material https://gitlab.com/aigarashi/ReFX.

Code availability https://gitlab.com/aigarashi/ReFX.

Declarations

Conflict of interest No conflicts.

References

1. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. ACM Trans. Program. Lang. Syst. (2011). https://doi.org/10.1145/1890028.1890031
2. Benton, N.: A typed, compositional logic for a stack-based abstract machine. In: Proceedings of Asian Symposium on Programming Languages and Systems (APLAS), pp. 364–380. Springer, Berlin (2005). https://doi.org/10.1007/11575467_24
3. Bernardo, B., Cauderlier, R., Hu, Z., Pesin, B., Tesson, J.: 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, Lecture Notes in Computer Science, vol. 12232, pp. 368–379. Springer (2019). https://doi.org/10.1007/978-3-030-54994-7_28
4. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings, pp. 337–340 (2008).https://doi.org/10.1007/978-3-540-78800-3_24
5. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press, New York (2001)
6. Goodman, L.: Tezos—a self-amending crypto-ledger. white paper (2014). https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf. Accessed 14 Oct 2020
7. Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y.: Online detection of effectively callback free objects with applications to smart contracts. In: Proc. ACM Program. Lang., vol. 2 (POPL) (2017). https://doi.org/10.1145/3158136

8. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: KEVM: a complete formal semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), pp. 204–217 (2018). https://doi.org/10.1109/CSF.2018.00022
9. Hirai, Y.: Defining the Ethereum virtual machine for interactive theorem provers. In: Financial Cryptography and Data Security, pp. 520–535. Springer International Publishing (2017)
10. Honda, K., Yoshida, N.: A compositional logic for polymorphic higher-order functions. In: Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 24–26 August 2004, Verona, Italy, pp. 191–202. ACM (2004). https://doi.org/10.1145/1013963.1013985
11. Kawaguchi, M., Rondon, P.M., Jhala, R.: Type-based data structure verification. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15–21, 2009, pp. 304–315. ACM (2009). https://doi.org/10.1145/1542476.1542510
12. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4–8, 2011, pp. 222–233 (2011).https://doi.org/10.1145/1993498.1993525
13. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008). https://bitcoin.org/bitcoin.pdf. Accessed 12 Oct 2020
14. Nomadic Labs.: Michelson: the language of smart contracts in Tezos. https://tezos.gitlab.io/whitedoc/michelson.html. Accessed 14 Oct 2020
15. Owens, S., Böhm, P., Zappa Nardelli, F., Sewell, P.: Lem: a lightweight tool for heavyweight semantics. In: Interactive Theorem Proving, pp. 363–369. Springer, Berlin (2011)
16. Park, D., Zhang, Y., Saxena, M., Daian, P., Roşu, G.: 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, pp. 912–915. ACM (2018). https://doi.org/10.1145/3236024.3264591
17. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7–13, 2008, pp. 159–169 (2008).https://doi.org/10.1145/1375581.1375602
18. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Logic Algebraic Program. **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
19. Siegel, D.: Understanding the DAO attack. CoinDesk (2016). https://www.coindesk.com/understanding-dao-hack-journalists. Accessed 13 Oct 2020
20. Szabo, N.: Formalizing and securing relationships on public networks. First Monday (1997). https://doi.org/10.5210/fm.v2i9.548
21. Terauchi, T.: Dependent types from counterexamples. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17–23, 2010, pp. 119–130 (2010). https://doi.org/10.1145/1706299.1706315
22. The Coq development team.: The coq proof assistant reference manual. Version 8.12.0 (2020). http://coq.inria.fr
23. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7–9, 2009, Coimbra, Portugal, pp. 277–288 (2009). https://doi.org/10.1145/1599410.1599445
24. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Jones, S.L.P.: Refinement types for Haskell. In: Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1–3, 2014, pp. 269–282. ACM (2014). https://doi.org/10.1145/2628136.2628161
25. Vazou, N., Tondwalkar, A., Choudhury, V., Scott, R.G., Newton, R.R., Wadler, P., Jhala, R.: Refinement reflection: Complete verification with SMT. In: Proc. ACM Program. Lang., vol. 2 (POPL) (2017). https://doi.org/10.1145/3158141
26. Xi, H.: Dependent ML an approach to practical programming with dependent types. J. Funct. Program. **17**(2), 215–286 (2007). https://doi.org/10.1017/S0956796806006216
27. Xi, H., Harper, R.: A dependently typed assembly language. In: Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3–5, 2001, pp. 169–180. ACM (2001). https://doi.org/10.1145/507635.507657

28. Zhu, H., Jagannathan, S.: Compositional and lightweight dependent type inference for ML. In: Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20–22, 2013. Proceedings, pp. 295–314 (2013). https://doi.org/10.1007/978-3-642-35873-9_19

Authors and Affiliations

Yuki Nishida[1] · Hiromasa Saito[1] · Ran Chen[1] · Akira Kawata[1] · Jun Furuse[2] · Kohei Suenaga[1] · Atsushi Igarashi[1]

Hiromasa Saito
hsaito@fos.kuis.kyoto-u.ac.jp

Ran Chen
aran@fos.kuis.kyoto-u.ac.jp

Akira Kawata
akira@fos.kuis.kyoto-u.ac.jp

Jun Furuse
jun.furuse@dailambda.jp

Kohei Suenaga
ksuenaga@fos.kuis.kyoto-u.ac.jp

Atsushi Igarashi
igarashi@fos.kuis.kyoto-u.ac.jp

[1] Kyoto University, Kyoto, Japan

[2] DaiLambda, Inc., Kyoto, Japan