Formal Verification of Deed Contract in Ethereum Name Service

Yoichi Hirai[*]

November 1, 2016

Contents

1 Introduction **3**
- 1.1 What this Document is . . . 3
- 1.2 Which Smart Contract this Document is about . . . 3
- 1.3 What is Proven . . . 3
- 1.4 What can Go Wrong . . . 4
- 1.5 Verifying Other Contracts . . . 5
- 1.6 Links . . . 5

2 Some Data Types for EVM **5**

3 EVM Instructions **7**
- 3.1 Bit Operations . . . 8
- 3.2 Signed Arithmetics . . . 8
- 3.3 Unsigned Arithmetics . . . 9
- 3.4 Informational Instructions . . . 9
- 3.5 Duplicating Stack Elements . . . 10
- 3.6 Memory Operations . . . 11
- 3.7 Storage Operations . . . 11
- 3.8 Program-Counter Instructions . . . 11
- 3.9 Stack Instructions . . . 12
- 3.10 Logging Instructions . . . 12
- 3.11 Miscellaneous Instructions . . . 13
- 3.12 Annotation Instruction . . . 13
- 3.13 The Whole Instruction Set . . . 13

[*] i@yoichihirai.com

4 A Contract Centric View of the EVM 15
4.1 Utility Functions 15
4.2 The Interaction between the Contract and the World 16
4.2.1 The World's Moves 16
4.2.2 The Contract's Moves 17
4.3 Program Representation 18
4.4 Translating an Instruction List into a Program 18
4.4.1 Integers can be compared 18
4.4.2 Storing the immediate values in the AVL tree 19
4.4.3 Storing a program in the AVL tree 19
4.4.4 Storing annotations in a program in a mapping 20
4.4.5 Translating a list of instructions into a program 20
4.5 Program as a Byte Sequence 20
4.6 Execution Environments 21
4.7 The Result of an Instruction 21
4.8 Useful Functions for Defining EVM Operations 22
4.9 Definition of EVM Operations 25
4.10 Programs' Answer to the World 38
4.11 Account's State 40
4.12 Environment Construction before EVM Execution 40
4.13 Account State Update after EVM Execution 44
4.14 Controlling the Isabelle Simplifier 45

5 What can Happen during a Contract Invocation 48
5.1 Some Possible Changes on Our Account State 49
5.2 A Round of the Game 51
5.3 Repetitions of rounds 53
5.4 How to State an Invariant 54
5.5 How to State a Pre-Post Condition Pair 55

6 Verification of the Deed Contract 57
6.1 The Code under Verification 57
6.2 The Invariant 68
6.3 Proof that the Invariant is Kept 69
6.4 Proof about the Case when the Caller is Not the Registrar . . 74

1 Introduction

1.1 What this Document is

This document describes a formal verification result about one contract in an Ethereum Name Service implementation. This document was generated by the Isabelle proof assistant[1]. Isabelle/HOL checked all lemmata to be correct (though it cannot check the definitions against reality).

The verified contract is relatively small, but this is the first "real"[2] contract that I have analyzed in a theorem proving environment.

The verification is result far from perfect. I am still finding more problems in the verification setup than in the verified contracts. The EVM (Ethereum Virtual Machine) implementation is not tested against others!

I am making this public already because this project makes a good example on the amount of work (and the level of detail) required to verify a smart contract using the machine-assisted logical inference. At this point already, if I were to implement a smart contract that holds more than 100k dollars, and if I am in charge of the schedule, I would consider this kind of development (the other option is to try the contract with smaller values first).

Acknowledgment. Most of the work has been carried out during my working hours in Ethereum DEV UG. I would like to thank Sami Mäkelä for suggestions and corrections of my mistakes.

1.2 Which Smart Contract this Document is about

The target of the verification is the `Deed` contract[3], which is a part of the Ethereum Name Service. The current development uses the bytecode obtained from the Solidity compiler version https://github.com/ethereum/solidity/tree/2d9109ba453d49547778c39a506b0ed492305c16.

Most parts of this document are generic to all Ethereum smart contracts. Only the last section is specific to the Deed contract.

1.3 What is Proven

The proven property is about one invocation of the Deed contract. In short, "only the registrar can decrease the balance." The invocation can be deep

[1] However, that is not the guarantee of truth. Check the notion of "Pollack inconsistency".

[2] The word "real" means it is aimed for production.

[3] https://github.com/Arachnid/ens/blob/f3334337083728728da56824a5d0a30a8712b60c/HashRegistrarSimplified.sol#L25

into the nested reentrancy calls, but the property holds for any one of these. I can list the assumptions and the implied conclusions.

Assumptions at the invocation:

- the account has the bytecode of the Deed contract or the account has no code;
- the caller's address it not the one stored at index 0 of the account's storage (i.e. the caller is not the `registrar`)[4];
- the 21st least byte in storage index 2 is not zero (i.e. the contract is still `active`)[5];
- the account's balance and the sent value added together do not overflow the range of 256-bit unsigned integers;
- the account is not marked as killed.

Conclusions when the invocation finishes (i.e. when the contract returns or fails back to the same depth in the callstack):

- the account's balance after the call is not smaller than the account's balance before the call;
- the account is not marked as killed after the call; and
- the 21st least byte in storage index 2 is still not zero (the account is still `active`);
- the storage content at index 0 (the `registrar`) is not changed.

1.4 What can Go Wrong

The property above is only a safety property; it states something bad is not going to happen. This does not mean anything good happens. While it is feasible to prove that something good happens, at least I need to model the gas mechanism in Isabelle/HOL before I claim anything works. In any case, when you have a sequence of events in mind, you can test that. Theorem proving excels at uncovering unknown possibilities.

The analysis cuts corners but it is reasonably equipped for safety properties. The analysis considers reentrancy, the account erasure after execution of the `SUICIDE` opcode, the byte level organization of EVM memory and storage,

[4] I'm guessing that the storage index 0 contains the registrar's address. To do this properly I need the help of the Solidity compiler.

[5] I didn't know this condition until Isabelle/HOL complained. This shows the possibility of using theorem provers to find vulnerabilities.

and the fact that the balance of an account can increase even when the code of the account is not invoked. The analysis is not aware of out-of-gas failures and the stack depth failures, but the analysis does not miss any kind of account state changes because of that. The verification is given up at the moment DELEGATE or CALLCODE instruction is used.

The hex output from Solidity is parsed by a new parser, which might produce incorrect results.

The biggest pitfall currently is the untested EVM implementation. Although this is a new EVM implementation, it is not tested against the standard EVM tests! This is already wrong. So, after getting this document in shape, the next thing I try is to test the new EVM implementation against the standard EVM tests (but before this I need to implement the gas). There is one good thing about the new EVM implementation. It allows us to reason about all possible executions!

1.5 Verifying Other Contracts

Most parts of the development can be reused for verifying other contracts. The amount of work is different for every smart contract. It would be relatively straightforward to do something similar for contracts without loops. For contracts with loops, either lots of manual work or some more sophisticated machinery is necessary.

This development so far does not use sophisticated techniques. An appropriate description would be "a brute-force approach based on a bare machine model." Apart from the tool Isabelle/HOL itself, I used no verification techniques from the 21st century yet.

1.6 Links

- This document is produced from the code available at https://github.com/pirapira/eth-isabelle/tree/deed.
- To get updates on this project and similar ones, follow http://gitter.im/ethereum/formal-methods.

2 Some Data Types for EVM

This development depends on Isabelle/HOL's machine word library. The machine word library is one of the biggest reasons for choosing Isabelle/HOL for this development. The Ethereum Virtual Machine depends on 8-bit bytes and 256-bit machine words.

theory *ContractEnv*

imports *Main* ~~/*src*/*HOL*/*Word*/*Word*

begin

The frequently used machine word types are named here. For example, *address* is the type of 160-bit machine words. The type *w256* is the type of EVM machine words.

type-synonym *w256* = *256 word* — 256 bit words
type-synonym *address* = *160 word* — 160 bit addresses
type-synonym *byte* = *8 word* — 8 bit bytes

In EVM, the memory contains one byte for each machine word (offset). The storage contains one machine word for each machine word (index). As we will see, the memory is cleared for every invocation of smart contracts. The storage is persistent for an account.

type-synonym *memory* = *w256* ⇒ *byte*
type-synonym *storage* = *w256* ⇒ *w256*

The storage is modelled as a function. For example, the empty storage is a function that returns zero for every index. Initially all accounts come with the empty storage.

definition *empty-storage* :: *storage*
where
empty-storage = (λ -. *0*)

During proofs, the definition of *empty-storage* is expanded automatically.

declare *empty-storage-def* [*simp*]

The empty memory is very similar.

definition *empty-memory* :: *memory*
where
empty-memory = (λ -. *0*)

declare *empty-memory-def* [*simp*]

The following record lists the information available for bytecode-inline assertions. These assertions will be proved in Isabelle/HOL.

record *aenv* =
  aenv-stack :: *w256 list* — the current stack
  aenv-memory :: *memory* — the current memory
  aenv-storage :: *storage* — the current storage
  aenv-balance :: *address* ⇒ *w256* — the current balance of all accounts
  aenv-caller :: *address* — the caller of the current invocation
  aenv-value-sent :: *w256* — the amount of Eth sent alont the current invocation
  aenv-data-sent :: *byte list* — the data sent along the current invocation
  aenv-storage-at-call :: *storage* — the storage content at the time of the invocation

aenv-balance-at-call :: *address* ⇒ *w256*
— the balance of all accounts at the time of the invocation
aenv-this :: *address* — the address of this contract under verification
aenv-origin :: *address* — the external account that started the transaction.

aenv-balance field keeps track of the balance of all accounts because the contract under verification can send some Eth to other accounts. To capture the effect of this, I chose to keep track of the balances of the other contracts.

aenv-storage-at-call and *aenv-balance-at-call* fields remember the states at the time of the contract invocation. These are used for rolling back the state after a failure. Failures happen for example when the contract under verification jumps to a wrong destination, or it runs out of gas.

aenv-origin might be the same as but might be different from *aenv-caller*. An Ethereum transaction is started by an external account (that is, an account which does not have codes but owned by somebody with a secret key). *aenv-origin* denotes this external account. During a transaction, the origin first sends a message to an account, the receiver can in turn call other accounts as well. When the calls nest, *aenv-caller* points to the immediate caller of the current invocation.

I'm going to add more fields in the *aenv* record in the near future because it does not contain all the information available at the execution time.

end

3 EVM Instructions

This section lists the EVM instructions and their byte representations. I also introduce an assertion instruction, whose byte representation is empty. The assertion instruction is a statement about the state of the EVM at that position of the program.

In Isabelle/HOL, it is expensive to define a single inductive type that contains all instructions. When I do it, Isabelle/HOL automatically proves every instruction is different from any other instruction, but this process has the computational complexity of the square of the number of instructions. Instead, I define multiple smaller inductive types and unify them at the end.

theory *Instructions*

imports *Main ~~/src/HOL/Word/Word ./ContractEnv*

begin

3.1 Bit Operations

The following clause defines a type called *bits_inst*. The type has five elements. It is automatically understood that nothing else belongs to this type. It is also understood that every one of these five elements is different from any of the other four.

Some instructions have *inst_* in front because names like AND, OR and XOR are taken by the machine word library.

The instructions have different arities. They might consume some elements on the stack, and produce some elements on the stack. However, the arity of the instructions are not specified in this section.

datatype *bits-inst*
= *inst-AND* — bitwise AND
| *inst-OR* — bitwise OR
| *inst-XOR* — bitwise exclusive or
| *inst-NOT* — bitwise negation
| *BYTE* — taking one byte out of a word

These instructions are represented by the following bytes. Most opcodes are a single byte.

fun *bits-inst-code* :: *bits-inst* ⇒ *byte*
where
bits-inst-code inst-AND = *0x16*
| *bits-inst-code inst-OR* = *0x17*
| *bits-inst-code inst-XOR* = *0x18*
| *bits-inst-code inst-NOT* = *0x19*
| *bits-inst-code BYTE* = *0x1a*

declare *bits-inst-code.simps* [*simp*]

3.2 Signed Arithmetics

More similar definitions follow. Below are instructions for signed arithmetics. The operations common to signed and unsigned are listed further below in the Unsigned Arithmetics section.

datatype *sarith-inst*
= *SDIV* — signed division
| *SMOD* — signed modulo
| *SGT* — signed greater-than
| *SLT* — signed less-than
| *SIGNEXTEND* — extend the size of a signed number

fun *sarith-inst-code* :: *sarith-inst* => *byte*
where
sarith-inst-code SDIV = *0x05*

| *sarith-inst-code SMOD = 0x07*
| *sarith-inst-code SGT = 0x13*
| *sarith-inst-code SLT = 0x12*
| *sarith-inst-code SIGNEXTEND = 0x0b*

declare *sarith-inst-code.simps* [*simp*]

3.3 Unsigned Arithmetics

The names GT, EQ and LT are taken in the Cmp library (which will be used for AVL trees).

datatype *arith-inst*
= *ADD* — addition
| *MUL* — multiplication
| *SUB* — subtraction
| *DIV* — unsigned division
| *MOD* — unsigned modulo
| *ADDMOD* — addition under modulo
| *MULMOD* — multiplication under modulo
| *EXP* — exponentiation
| *inst-GT* — unsigned greater-than
| *inst-EQ* — equality
| *inst-LT* — unsigned less-than
| *ISZERO* — if zero, returns one
| *SHA3* — Keccak 256, dispite the name

fun *arith-inst-code* :: *arith-inst* ⇒ *byte*
where
arith-inst-code ADD = 0x01
| *arith-inst-code MUL = 0x02*
| *arith-inst-code SUB = 0x03*
| *arith-inst-code DIV = 0x04*
| *arith-inst-code MOD = 0x06*
| *arith-inst-code ADDMOD = 0x08*
| *arith-inst-code MULMOD = 0x09*
| *arith-inst-code EXP = 0x0a*
| *arith-inst-code inst-GT = 0x11*
| *arith-inst-code inst-LT = 0x10*
| *arith-inst-code inst-EQ = 0x14*
| *arith-inst-code ISZERO = 0x15*
| *arith-inst-code SHA3 = 0x20*

declare *arith-inst-code.simps* [*simp*]

3.4 Informational Instructions

datatype *info-inst* =
ADDRESS — the address of the account currently running
| *BALANCE* — the Eth balance of the specified account

| *ORIGIN* — the address of the external account that started the transaction
| *CALLER* — the immediate caller of this invocation
| *CALLVALUE* — the Eth amount sent along this invocation
| *CALLDATASIZE* — The number of bytes sent along this invocation
| *CODESIZE* — the number of bytes in the currently running code
| *GASPRICE* — the current gas price
| *EXTCODESIZE* — the size of the code on the specified account
| *BLOCKHASH* — the block hash of a specified block among the recent blocks
| *COINBASE* — the address of the miner that validates the current block
| *TIMESTAMP* — the date and time of the block
| *NUMBER* — the block number
| *DIFFICULTY* — the current difficulty
| *GASLIMIT* — the current block gas limit
| *GAS* — the remaining gas for the current execution.

fun *info-inst-code* :: *info-inst* ⇒ *byte*
where
info-inst-code ADDRESS = 0x30
| *info-inst-code BALANCE = 0x31*
| *info-inst-code ORIGIN = 0x32*
| *info-inst-code CALLVALUE = 0x34*
| *info-inst-code CALLDATASIZE = 0x36*
| *info-inst-code CALLER = 0x33*
| *info-inst-code CODESIZE = 0x38*
| *info-inst-code GASPRICE = 0x3a*
| *info-inst-code EXTCODESIZE = 0x3b*
| *info-inst-code BLOCKHASH = 0x40*
| *info-inst-code COINBASE = 0x41*
| *info-inst-code TIMESTAMP = 0x42*
| *info-inst-code NUMBER = 0x43*
| *info-inst-code DIFFICULTY = 0x44*
| *info-inst-code GASLIMIT = 0x45*
| *info-inst-code GAS = 0x5a*

declare *info-inst-code.simps* [*simp*]

3.5 Duplicating Stack Elements

There are sixteen instructions for duplicating a stack element. These instructions take a stack element and duplicate it on top of the stack.

type-synonym *dup-inst* = *nat*

abbreviation *dup-inst-code* :: *dup-inst* ⇒ *byte*
where
dup-inst-code n ≡
(*if n < 1 then undefined* (∗ *There is no DUP0 instruction.* ∗)
else (*if n > 16 then undefined* (∗ *There are no DUP16 instruction and on.* ∗)
else (*word-of-int* (*int n*)) + *0x7f*))
— 0x80 stands for DUP1 until 0x9f for DUP16.

3.6 Memory Operations

datatype *memory-inst* =
  MLOAD — reading one word from the memory from the specified offset
  | *MSTORE* — writing one machine word to the memory
  | *MSTORE8* — writing one byte to the memory
  | *CALLDATACOPY* — copying the caller's data to the memory
  | *CODECOPY* — copying a part of the currently running code to the memory
  | *EXTCODECOPY* — copying a part of the code of the specified account
  | *MSIZE* — the size of the currently used region of the memory.

fun *memory-inst-code* :: *memory-inst* ⇒ *byte*
where
  memory-inst-code MLOAD = *0x51*
| *memory-inst-code MSTORE* = *0x52*
| *memory-inst-code MSTORE8* = *0x53*
| *memory-inst-code CALLDATACOPY* = *0x37*
| *memory-inst-code CODECOPY* = *0x39*
| *memory-inst-code EXTCODECOPY* = *0x3c*
| *memory-inst-code MSIZE* = *0x59*

declare *memory-inst-code.simps* [*simp*]

3.7 Storage Operations

datatype *storage-inst* =
  SLOAD — reading one word from the storage
  | *SSTORE* — writing one word to the storage.

fun *storage-inst-code* :: *storage-inst* ⇒ *byte*
where
  storage-inst-code SLOAD = *0x54*
| *storage-inst-code SSTORE* = *0x55*

declare *storage-inst-code.simps* [*simp*]

3.8 Program-Counter Instructions

datatype *pc-inst* =
  JUMP — jumping to the specified location in the code
  | *JUMPI* — jumping to the specified location in the code if a condition is met
  | *PC* — the current location in the code
  | *JUMPDEST* — a no-op instruction located to indicate jump destinations.

If a jump occurs to a location where *JUMPDEST* is not found, the execution fails.

fun *pc-inst-code* :: *pc-inst* ⇒ *byte*
where
  pc-inst-code JUMP = *0x56*
| *pc-inst-code JUMPI* = *0x57*

| *pc-inst-code PC* = *0x58*
| *pc-inst-code JUMPDEST* = *0x5b*

declare *pc-inst-code.simps* [*simp*]

3.9 Stack Instructions

datatype *stack-inst* =
POP — throwing away the topmost element of the stack
| *PUSH-N 8 word list* — pushing an element to the stack
| *CALLDATALOAD* — pushing a word to the stack, taken from the caller's data.

The PUSH instructions have longer byte representations than the other instructions because they contain immediate values. Here the immediate value is represented by a list of bytes. Depending on the length of the list, the PUSH operation takes different opcodes.

fun *stack-inst-code* :: *stack-inst* ⇒ *byte list*
where
stack-inst-code POP = [*0x50*]
| *stack-inst-code* (*PUSH-N lst*) =
(*if* (*size lst*) < *1 then undefined* (∗ *there is no PUSH0 instruction* ∗)
else (*if* (*size lst*) > *32 then undefined* (∗ *there are no PUSH33 and so on* ∗)
else word-of-int (*int* (*size lst*)) + *0x5f*)) # *lst*
| *stack-inst-code CALLDATALOAD* = [*0x35*]

declare *stack-inst-code.simps* [*simp*]

type-synonym *swap-inst* = *nat*

abbreviation *swap-inst-code* :: *swap-inst* ⇒ *byte*
where
swap-inst-code n ≡
(*if n* < *1 then undefined else* (∗ *there is no SWAP0* ∗)
(*if n* > *16 then undefined else* (∗ *there are no SWAP17 and on* ∗)
word-of-int (*int n*) + *0x8f*))

3.10 Logging Instructions

There are instructions for logging events with different number of arguments.

datatype *log-inst* = *LOG0* | *LOG1* | *LOG2* | *LOG3* | *LOG4*

fun *log-inst-code* :: *log-inst* ⇒ *byte*
where
log-inst-code LOG0 = *0xa0*
| *log-inst-code LOG1* = *0xa1*
| *log-inst-code LOG2* = *0xa2*
| *log-inst-code LOG3* = *0xa3*
| *log-inst-code LOG4* = *0xa4*

declare *log-inst-code.simps* [*simp*]

3.11 Miscellaneous Instructions

This section contains the instructions that alter the account-wise control flow. In other words, they cause communication between accounts (or at least interaction with other accounts' code).

datatype *misc-inst*
= *STOP* — finishing the execution normally, with the empty return data
| *CREATE* — deploying some code in an account
| *CALL* — calling (i.e. sending a message to) an account
| *CALLCODE* — calling into the current account with some other account's code

| *DELEGATECALL*
— calling into this account, the executed code can be some other account's but the sent value and the sent data are unchanged.
| *RETURN* — finishing the execution normally with data
| *SUICIDE*
— send all remaining Eth balance to the specified account, finishing the execution normally, and flagging the current account for deletion.

fun *misc-inst-code* :: *misc-inst* ⇒ *byte*
where
misc-inst-code STOP = *0x00*
| *misc-inst-code CREATE* = *0xf0*
| *misc-inst-code CALL* = *0xf1*
| *misc-inst-code CALLCODE* = *0xf2*
| *misc-inst-code RETURN* = *0xf3*
| *misc-inst-code DELEGATECALL* = *0xf4*
| *misc-inst-code SUICIDE* = *0xff*

declare *misc-inst-code.simps* [*simp*]

3.12 Annotation Instruction

The annotation instruction is just a predicate over *aenv*. A predicate is modelled as a function returning a boolean.

type-synonym *annotation* = *aenv* ⇒ *bool*

3.13 The Whole Instruction Set

The small inductive sets above are here combined into a single type.

datatype *inst* =
Unknown byte
| *Bits bits-inst*
| *Sarith sarith-inst*

| *Arith arith-inst*
| *Info info-inst*
| *Dup dup-inst*
| *Memory memory-inst*
| *Storage storage-inst*
| *Pc pc-inst*
| *Stack stack-inst*
| *Swap swap-inst*
| *Log log-inst*
| *Misc misc-inst*
| *Annotation annotation*

And the byte representation of these instructions are defined.

fun *inst-code* :: *inst* ⇒ *byte list*
where
inst-code (*Unknown byte*) = [*byte*]
| *inst-code* (*Bits b*) = [*bits-inst-code b*]
| *inst-code* (*Sarith s*) = [*sarith-inst-code s*]
| *inst-code* (*Arith a*) = [*arith-inst-code a*]
| *inst-code* (*Info i*) = [*info-inst-code i*]
| *inst-code* (*Dup d*) = [*dup-inst-code d*]
| *inst-code* (*Memory m*) = [*memory-inst-code m*]
| *inst-code* (*Storage s*) = [*storage-inst-code s*]
| *inst-code* (*Pc p*) = [*pc-inst-code p*]
| *inst-code* (*Stack s*) = *stack-inst-code s*
| *inst-code* (*Swap s*) = [*swap-inst-code s*]
| *inst-code* (*Log l*) = [*log-inst-code l*]
| *inst-code* (*Misc m*) = [*misc-inst-code m*]
| *inst-code* (*Annotation -*) = []

declare *inst-code.simps* [*simp*]

The size of an opcode is useful for parsing a hex representation of an EVM code.

abbreviation *inst-size* :: *inst* ⇒ *int*
where
inst-size i ≡ *int* (*length* (*inst-code i*))

This can also be used to find jump destinations from a sequence of opcodes.

fun *drop-bytes* :: *inst list* ⇒ *nat* ⇒ *inst list*
where
drop-bytes prg 0 = *prg*
| *drop-bytes* (*Stack* (*PUSH-N v*) # *rest*) *bytes* =
drop-bytes rest (*bytes* − *1* − *length v*)
| *drop-bytes* (*Annotation -* # *rest*) *bytes* = *drop-bytes rest bytes*
| *drop-bytes* (*-* # *rest*) *bytes* = *drop-bytes rest* (*bytes* − *1*)
| *drop-bytes* [] (*Suc v*) = []

declare *drop-bytes.simps* [*simp*]

Also it is possible to compute the size of a program as the number of bytes,

fun *program-size* :: *inst list* ⇒ *nat*
where
  program-size (*Stack* (*PUSH-N v*) # *rest*) = *length v* + *1* + *program-size rest*
  — I was using *inst-size* here, but that contributed to performance problems.
| *program-size* (*Annotation* - # *rest*) = *program-size rest*
| *program-size* (- # *rest*) = *1* + *program-size rest*
| *program-size* [] = *0*

declare *program-size.simps* [*simp*]

as well as computing the byte representation of the program.

fun *program-code* :: *inst list* ⇒ *byte list*
where
  program-code [] = []
| *program-code* (*inst* # *rest*) = *inst-code inst* @ *program-code rest*

declare *program-code.simps* [*simp*]

end

4 A Contract Centric View of the EVM

Here is a presentation of the Ethereum Virtual Machine (EVM) in a form suitable for formal verification of a single account.

theory *ContractSem*

imports *Main* *~~/src/HOL/Word/Word* *~~/src/HOL/Data-Structures/AVL-Map* *./ContractEnv* *./Instructions* *./KEC*

begin

4.1 Utility Functions

The following function is an if-sentence, but with some strict control over the evaluation order. Neither the then-clause nor the else-clause is simplified during proofs. This prevents the automatic simplifier from computing the results of both the then-clause and the else-clause.

definition *strict-if* :: *bool* ⇒ (*bool* ⇒ *'a*) ⇒ (*bool* ⇒ *'a*) ⇒ *'a*
where
strict-if b x y = (*if b then x True else y True*)

When the if-condition is known to be True, the simplifier can proceed into the then-clause. The *simp* attribute encourages the simplifier to use this equation from left to right whenever applicable.

lemma *strict-if-True* [*simp*] :

strict-if True a b = *a True*
apply(*simp add*: *strict-if-def*)
done

When the if-condition is known to be False, the simplifier can proceed into the else-clause.

lemma *strict-if-False* [*simp*] :
strict-if False a b = *b True*
apply(*simp add*: *strict-if-def*)
done

When the if-condition is not known to be either True or False, the simplifier is allowed to perform computation on the if-condition. The *cong* attribute tells the simplifier to try to rewrite the left hand side of the conclusion, using the assumption.

lemma *strict-if-cong* [*cong*] :
b0 = *b1* $\Longrightarrow$ *strict-if b0 x y* = *strict-if b1 x y*
apply(*auto*)
done

4.2 The Interaction between the Contract and the World

In this development, the EVM execution is seen as an interaction between a single contract invocation and the rest of the world. The world can call into the contract. The contract can reply by just finishing or failing, but it can also call an account[6]. When our contract execution calls an account, this is seen as an action towards the world, because the world then has to decide the result of this call. The world can say that the call finished successfully or exceptionally. The world can also say that the call resulted in a reentrancy. In other words, the world can call the contract again and change the storage and the balance of our contract. The whole process is captured as a game between the world and the contract.

4.2.1 The World's Moves

The world can call into our contract. Then the world provides our[7] contract with the following information.

record *call-env* =
  callenv-gaslimit :: *w256* — the current block's gas limit
  callenv-value :: *w256* — the amount of Eth sent along
  callenv-data :: *byte list* — the data sent along

[6]This might be the same account as our invocation, but still the deeper calls is part of the world.

[7] The contract's behavior is controlled by a concrete code, but the world's behavior is unrestricted. So when I get emotional I call the contract "our" contract.

callenv-caller :: *address* — the caller's address
callenv-timestamp :: *w256* — the timestamp of the current block
callenv-blocknum :: *w256* — the block number of the current block
callenv-balance :: *address* ⇒ *w256* — the balances of all accounts.

After our contract calls accounts, the world can make those accounts return into our contracts. The return value is not under control of our current contract, so it is the world's move. In that case, the world provides the following information.

record *return-result* =
return-data :: *byte list* — the returned data
return-balance :: *address* ⇒ *w256*
— the balance of all accounts at the moment of the return

Even our account's balance (and its storage) might have changed at this moment. *return-result* type is also used when our contract returns, as we will see.

With these definitions now we can define the world's actions. In addition to call and return, there is another clause for failing back to the account. This happens when our contract calls an account but the called account fails.

datatype *world-action* =
WorldCall call-env — the world calls into the account
| *WorldRet return-result* — the world returns back to the account
| *WorldFail* — the world fails back to the account.

4.2.2 The Contract's Moves

After being invoked, the contract can respond by calling an account, creating (or deploying) a smart contract, destroying itself, returning, or failing. When the contract calls an account, the contract provides the following information.

record *call-arguments* =
callarg-gas :: *w256* — the portion of the remaining gas that the callee is allowed to use
callarg-code :: *address* — the code that executes during the call
callarg-recipient :: *address* — the recipient of the call, whose balance and the storage are modified.
callarg-value :: *w256* — the amount of Eth sent along
callarg-data :: *byte list* — the data sent along
callarg-output-begin :: *w256* — the beginning of the memory region where the output data should be written.
callarg-output-size :: *w256* — the size of the memory regions where the output data should be written.

When our contract deploys a smart contract, our contract should provide the following information.

record *create-arguments* =
  createarg-value :: *w256* — the value sent to the account
  createarg-code :: *byte list* — the code that deploys the runtime code.

The contract's moves are summarized as follows.

datatype *contract-action* =
  ContractCall call-arguments — calling an account
| *ContractCreate create-arguments* — deploying a smart contract
| *ContractFail* — failing back to the caller
| *ContractSuicide* — destroying itself and returning back to the caller
| *ContractReturn byte list* — normally returning back to the caller

4.3 Program Representation

For performance reasons, the instructions are stored in an AVL tree that allows looking up instructions from the program counters.

record *program* =
  program-content :: (*int* × *inst*) *avl-tree*
  — a binary search tree that allows looking up instructions from positions
  program-length :: *int* — the length of the program in bytes
  program-annotation :: *int* ⇒ *annotation list*
  — a mapping from positions to annotations

The empty program is easy to define.

abbreviation *empty-program* :: *program*
where
empty-program ≡
  (| *program-content* = ⟨⟩
  , *program-length* = *0*
  , *program-annotation* = (λ -. []) |)

4.4 Translating an Instruction List into a Program

4.4.1 Integers can be compared

The AVL library requires the keys to be comparable. We represent program positions by integers. So we have to prove that integers belong to the type class *cmp* with the usual comparison operators.

instantiation *int* :: *cmp*
begin
definition *cmp-int* :: *int* ⇒ *int* ⇒ *Cmp.cmp*
where
cmp-int-def : *cmp-int x y* =
  (*if x* < *y then Cmp.LT else* (*if x* = *y then Cmp.EQ else Cmp.GT*))

instance proof
 fix *x y* :: *int* **show** (*cmp x y* = *cmp.LT*) = (*x* < *y*)

apply(*simp add*: *cmp-int-def*)
done
fix *x y* :: *int* **show** (*cmp x y* = *cmp.EQ*) = (*x* = *y*)
apply(*simp add*: *cmp-int-def*)
done
fix *x y* :: *int* **show** (*cmp x y* = *cmp.GT*) = (*x* > *y*)
apply(*simp add*: *cmp-int-def*)
done
qed

end

4.4.2 Storing the immediate values in the AVL tree

The data region of PUSH_N instructions are encoded as Unknown instructions. Here is a utility function that inserts a byte sequence after a specified index in the AVL tree.

fun *store-byte-list-in-program* ::
int (∗ *initial position in the AVL* ∗) ⇒ *byte list* (∗ *the data* ∗) ⇒
(*int* ∗ *inst*) *avl-tree* (∗ *original AVL* ∗) ⇒
(*int* ∗ *inst*) *avl-tree* (∗ *result* ∗)
where
store-byte-list-in-program - [] *orig* = *orig*
| *store-byte-list-in-program pos* (*h* # *t*) *orig* =
store-byte-list-in-program (*pos* + *1*) *t* (*update pos* (*Unknown h*) *orig*)
declare *store-byte-list-in-program.simps* [*simp*]

4.4.3 Storing a program in the AVL tree

Here is a function that stores a list of instructions in the AVL tree. The initial key is specified. The following keys are computed using the sizes of instructions being inserted.

fun *program-content-of-lst* ::
int (∗ *initial position in the AVL* ∗) ⇒ *inst list* (∗ *instructions* ∗)
⇒ (*int* ∗ *inst*) *avl-tree* (∗ *result* ∗)
where
program-content-of-lst - [] = *Leaf*
— the empty program is translated into the empty tree.
| *program-content-of-lst pos* (*Stack* (*PUSH-N bytes*) # *rest*) =
store-byte-list-in-program (*pos* + *1*) *bytes*
(*update pos* (*Stack* (*PUSH-N bytes*))
(*program-content-of-lst* (*pos* + *1* + (*int* (*length bytes*))) *rest*))
— The PUSH instruction is translated together with the immediate value.
| *program-content-of-lst pos* (*Annotation* - # *rest*) =
program-content-of-lst pos rest
— Annotations are skipped because they do not belong in this AVL tree.
| *program-content-of-lst pos* (*i* # *rest*) =
update pos i (*program-content-of-lst* (*pos* + *1*) *rest*)

— The other instructions are simply inserted into the AVL tree.

4.4.4 Storing annotations in a program in a mapping

Annotations are stored in a mapping that maps positions into lists of annotations. The rationale for this data structure is that a single position might contain multiple annotations. Here is a function that inserts an annotation at a specified position.

abbreviation *prepend-annotation* :: *int* ⇒ *annotation* ⇒ (*int* ⇒ *annotation list*) ⇒ (*int* ⇒ *annotation list*)
where
prepend-annotation pos annot orig ≡ *orig*(*pos* := *annot* # *orig pos*)

Currently annotations are inserted into a mapping with Isabelle/HOL's mapping updates. When this causes performance problems, I need to switch to AVL trees again.

fun *program-annotation-of-lst* :: *int* ⇒ *inst list* ⇒ *int* ⇒ *annotation list*
where
  program-annotation-of-lst - [] = (λ -. [])
| *program-annotation-of-lst pos* (*Annotation annot* # *rest*) =
    prepend-annotation pos annot (*program-annotation-of-lst pos rest*)
| *program-annotation-of-lst pos* (*i* # *rest*) =
   (*program-annotation-of-lst* (*pos* + *inst-size i*) *rest*)
  — Ordinary instructions are skipped.

declare *program-annotation-of-lst.simps* [*simp*]

4.4.5 Translating a list of instructions into a program

The results of the above translations are packed together in a record.

abbreviation *program-of-lst* :: *inst list* ⇒ *program*
where
program-of-lst lst ≡
  ⦇ *program-content* = *program-content-of-lst 0 lst*
  , *program-length* = *int* (*length lst*)
  , *program-annotation* = *program-annotation-of-lst 0 lst*
  ⦈

4.5 Program as a Byte Sequence

For CODECOPY instruction, the program must be seen as a byte-indexed read-only memory.

Such a memory is here implemented by a lookup on an AVL tree.

abbreviation *program-as-memory* :: *program* ⇒ *memory*
where
program-as-memory p idx ≡

(*case lookup* (*program-content p*) (*uint idx*) *of*
  None ⇒ *0*
| *Some inst* ⇒ *inst-code inst* ! *0*)

4.6 Execution Environments

I model an instruction as a function that takes environments and modifies some parts of them.

The execution of an EVM program happens in a block, and the following information about the block should be available.

record *block-info* =
  block-blockhash :: *w256* ⇒ *w256* — this captures the whole BLOCKHASH op
  block-coinbase :: *address* — the miner who validates the block
  block-timestamp :: *w256*
  block-number :: *w256* — the blocknumber of the block
  block-difficulty :: *w256*
  block-gaslimit :: *w256* — the block gas imit
  block-gasprice :: *w256*

The variable environment contains information that is relatively volatile.

record *variable-env* =
  venv-stack :: *w256 list*
  venv-memory :: *memory*
  venv-memory-usage :: *int* — the current memory usage
  venv-storage :: *storage*
  venv-pc :: *int* — the program counter
  venv-balance :: *address* ⇒ *w256* — balances of all accounts
  venv-caller :: *address* — the caller's address
  venv-value-sent :: *w256* — the amount of Eth sent along the current invocation
  venv-data-sent :: *byte list* — the data sent along the current invocation
  venv-storage-at-call :: *storage* — the storage content at the invocation
  venv-balance-at-call :: *address* ⇒ *w256* — the balances at the invocation
  venv-origin :: *address* — the external account that started the current transaction

  venv-ext-program :: *address* ⇒ *program* — the codes of all accounts
  venv-block :: *block-info* — the current block.

The constant environment contains information that is rather stable.

record *constant-env* =
  cenv-program :: *program* — the code in the account under verification
  cenv-this :: *address* — the address of the account under verification.

4.7 The Result of an Instruction

The result of program execution is microscopically defined by results of instruction executions. The execution of a single instruction can result in the following cases:

datatype *instruction-result* =
InstructionContinue variable-env — the execution should continue
| *InstructionAnnotationFailure* — the annotation turned out to be false
| *InstructionToWorld*
— the execution has stopped; either for the moment just calling out another account, or finally finishing the current invocation
contract-action (∗ *the contract's move* ∗)
× *storage* (∗ *the new storage content* ∗)
× (*address* ⇒ *w256*) (∗ *the new balance of all accounts* ∗)
× (*variable-env* × *int* × *int*) *option*
(∗ *the variable environment to return to*, ∗)
(∗ *and the memory reagion that expects the return value*. ∗)

When the contract fails, the result of the instruction always looks like this:

abbreviation *instruction-failure-result* :: *variable-env* ⇒ *instruction-result*
where
instruction-failure-result v ≡
InstructionToWorld
(*ContractFail*, *venv-storage-at-call v*, *venv-balance-at-call v*, *None*)

When the contract returns, the result of the instruction always looks like this:

abbreviation *instruction-return-result* :: *byte list* ⇒ *variable-env* ⇒ *instruction-result*
where
instruction-return-result x v ≡
InstructionToWorld (*ContractReturn x*, *venv-storage v*, *venv-balance v*, *None*)

4.8 Useful Functions for Defining EVM Operations

Currently the GAS instruction is modelled to return random numbers. The random number is not known to be of any value. However, the value is not unknown enough in this formalization because the value is only dependent on the variable environment (which does not keep track of the remaining gas). This is not a problem as long as we are analyzing a single invocation of a loopless contract, but gas accounting is a planned feature.

definition *gas* :: *variable-env* ⇒ *w256*
where *gas* - = *undefined*

This M function is defined at the end of H.1. in the yellow paper. This function is useful for updating the memory usage counter.

abbreviation M ::
int (∗ *original memory usage* ∗) ⇒ *w256* (∗ *beginning of the used memory* ∗)
⇒ *w256* (∗ *used size* ∗) ⇒ *int* (∗ *the updated memory usage* ∗)
where
M s f l ≡
(*if l* = *0 then s else*
max s ((*uint f* + *uint l* + *31*) *div 32*))

Updating a balance of a single account:

abbreviation *update-balance* ::
address (∗ *the updated account*∗)
⇒ (*w256* ⇒ *w256*) (∗ *the function that updates the balance* ∗)
⇒ (*address* ⇒ *w256*) (∗ *the original balance* ∗)
⇒ (*address* ⇒ *w256*) (∗ *the resulting balance* ∗)
where
update-balance a f orig ≡ *orig*(*a* := *f* (*orig a*))

Popping stack elements:

abbreviation *venv-pop-stack* ::
nat (∗ *how many elements to pop* ∗) ⇒ *variable-env* ⇒ *variable-env*
where
venv-pop-stack n v ≡
v⦇ *venv-stack* := *drop n* (*venv-stack v*) ⦈

Peeking the topmost element of the stack:

abbreviation *venv-stack-top* :: *variable-env* ⇒ *w256 option*
where
venv-stack-top v ≡
(*case venv-stack v of h* # -⇒ *Some h* | [] ⇒ *None*)

Updating the storage at an index:

abbreviation *venv-update-storage* ::
w256 (∗ *index* ∗) ⇒ *w256* (∗ *value* ∗)
⇒ *variable-env* (∗ *the original variable environment* ∗)
⇒ *variable-env* (∗ *the resulting variable environment* ∗)
where
venv-update-storage idx val v ≡
v⦇*venv-storage* := (*venv-storage v*)(*idx* := *val*)⦈

Peeking the next instruction:

abbreviation *venv-next-instruction* :: *variable-env* ⇒ *constant-env* ⇒ *inst option*
where
venv-next-instruction v c ≡
lookup (*program-content* (*cenv-program c*)) (*venv-pc v*)

Advancing the program counter:

abbreviation *venv-advance-pc* :: *constant-env* ⇒ *variable-env* ⇒ *variable-env*
where
venv-advance-pc c v ≡
v⦇ *venv-pc* := *venv-pc v* + *inst-size* (*the* (*venv-next-instruction v c*)) ⦈

No-op, which just advances the program counter:

abbreviation *stack-0-0-op* :: *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
stack-0-0-op v c ≡ *InstructionContinue* (*venv-advance-pc c v*)

A general pattern of operations that pushes one element onto the stack:

abbreviation *stack-0-1-op* ::
variable-env ⇒ *constant-env* ⇒ *w256* (∗ *the pushed word* ∗) ⇒ *instruction-result*
where
stack-0-1-op v c w ≡
InstructionContinue
(*venv-advance-pc c v*(|*venv-stack* := *w* # *venv-stack v*|))

A general pattern of operations that transforms the topmost element of the stack:

abbreviation *stack-1-1-op* :: *variable-env* ⇒ *constant-env* ⇒
(*w256* ⇒ *w256*) (∗ *the function that transforms a word*∗)
⇒ *instruction-result*
where
stack-1-1-op v c f ≡
(*case venv-stack v of*
[] ⇒ *instruction-failure-result v*
| *h* # *t* ⇒
InstructionContinue
(*venv-advance-pc c v*(|*venv-stack* := *f h* # *t*|)))

A general pattern of operations that consume one word and produce two rwords:

abbreviation *stack-1-2-op* ::
variable-env ⇒ *constant-env* ⇒ (*w256* ⇒ *w256* ∗ *w256*) ⇒ *instruction-result*
where
stack-1-2-op v c f ≡
(*case venv-stack v of*
[] ⇒ *instruction-failure-result v*
| *h* # *t* ⇒
(*case f h of*
(*new0*, *new1*) ⇒
InstructionContinue
(*venv-advance-pc c*
v(|*venv-stack* := *new0* # *new1* # *t* |))))

A general pattern of operations that take two words and produce one word:

abbreviation *stack-2-1-op* ::
variable-env ⇒ *constant-env* ⇒ (*w256* ⇒ *w256* ⇒ *w256*) ⇒ *instruction-result*
where
stack-2-1-op v c f ≡
(*case venv-stack v of*
operand0 # *operand1* # *rest* ⇒
InstructionContinue
(*venv-advance-pc c*
v(|*venv-stack* := *f operand0 operand1* # *rest*|))
| - ⇒ *instruction-failure-result v*)

A general pattern of operations that take three words and produce one word:

```
abbreviation stack-3-1-op ::
variable-env ⇒ constant-env ⇒
 (w256 ⇒ w256 ⇒ w256 ⇒ w256) ⇒ instruction-result
where
stack-3-1-op v c f ≡
  (case venv-stack v of
     operand0 # operand1 # operand2 # rest ⇒
       InstructionContinue
         (venv-advance-pc c
            v(|venv-stack := f operand0 operand1 operand2 # rest|))
   | - ⇒ instruction-failure-result v)
```

4.9 Definition of EVM Operations

SSTORE changes the storage so it does not fit into any of the patterns defined above.

```
abbreviation sstore :: variable-env ⇒ constant-env ⇒ instruction-result
where
sstore v c ≡
  (case venv-stack v of
    addr # val # stack-tail ⇒
      InstructionContinue
      (venv-advance-pc c
        (venv-update-storage addr val v(|venv-stack := stack-tail|)))
    | - ⇒ instruction-failure-result v)
```

For interpreting the annotations, I first need to construct the annotation environment out of the current execution environments. When I try to remove this step, I face some circular definitions of data types.

```
abbreviation build-aenv :: variable-env ⇒ constant-env ⇒ aenv
where
build-aenv v c ≡
  (| aenv-stack = venv-stack v
  , aenv-memory = venv-memory v
  , aenv-storage = venv-storage v
  , aenv-balance = venv-balance v
  , aenv-caller = venv-caller v
  , aenv-value-sent = venv-value-sent v
  , aenv-data-sent = venv-data-sent v
  , aenv-storage-at-call = venv-storage-at-call v
  , aenv-balance-at-call = venv-balance-at-call v
  , aenv-this = cenv-this c
  , aenv-origin = venv-origin v |)
```

In reality, EVM programs do not contain annotations so annotations never cause failures. However, during the verification, I want to catch annotation failures. When the annotation evaluates to False, the execution stops and results in *InstructionAnnotationFailure*.

definition *eval-annotation* :: *annotation* ⇒ *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
eval-annotation anno v c =
  (*if anno* (*build-aenv v c*)
   then
    InstructionContinue (*venv-advance-pc c v*)
   else
    InstructionAnnotationFailure)

The JUMP instruction has the following meaning. When it cannot find the JUMPDEST instruction at the destination, the execution fails.

abbreviation *jump* :: *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
jump v c ≡
 (*case venv-stack-top v of*
   None ⇒ *instruction-failure-result v*
  | *Some pos* ⇒
   (*let v-new* = (*venv-pop-stack* (*Suc 0*) *v*)(| *venv-pc* := *uint pos* |) *in*
   (*case venv-next-instruction v-new c of*
     Some (*Pc JUMPDEST*) ⇒
      InstructionContinue v-new
    | *Some* - ⇒ *instruction-failure-result v*
    | *None* ⇒ *instruction-failure-result v*)))

This function is a reminiscent of my struggle with the Isabelle/HOL simplifier. The second argument has no meaning but to control the Isabelle/HOL simplifier.

definition *blockedInstructionContinue* :: *variable-env* ⇒ *bool* ⇒ *instruction-result*
where
blockedInstructionContinue v - = *InstructionContinue v*

When the second argument is already *True*, the simplification can continue. Otherwise, the Isabelle/HOL simplifier is not allowed to expand the definition of *blockedInstructionContinue*.

lemma *unblockInstructionContinue* [*simp*] :
blockedInstructionContinue v True = *InstructionContinue v*
apply(*simp add*: *blockedInstructionContinue-def*)
done

This is another reminiscent of my struggle against the Isabelle/HOL simplifier. Again, the simplifier is not allowed to expand the definition unless the second argument is known to be *True*.

definition *blocked-jump* :: *variable-env* ⇒ *constant-env* ⇒ *bool* ⇒ *instruction-result*
where
blocked-jump v c - = *jump v c*

lemma *unblock-jump* [*simp*]:

blocked-jump v c True = jump v c
apply(*simp add*: *blocked-jump-def*)
done

The JUMPI instruction is implemented using the JUMP instruction.

abbreviation *jumpi* :: *variable-env ⇒ constant-env ⇒ instruction-result*
where
jumpi v c ≡
  (*case venv-stack v of*
    pos # cond # rest ⇒
      (*strict-if* (*cond = 0*)
        (*blockedInstructionContinue*
          (*venv-advance-pc c* (*venv-pop-stack* (*Suc* (*Suc 0*)) *v*)))
        (*blocked-jump* (*v*⦇ *venv-stack := pos # rest* ⦈) *c*))
   | - ⇒ *instruction-failure-result v*)

Looking up the call data size takes this work:

abbreviation *datasize* :: *variable-env ⇒ w256*
where
datasize v ≡ Word.word-of-int (*int* (*length* (*venv-data-sent v*)))

Looking up a word from a list of bytes:

abbreviation *read-word-from-bytes* :: *nat ⇒ byte list ⇒ w256*
where
read-word-from-bytes idx lst ==
  Word.word-rcat (*take 32* (*drop idx lst*))

Looking up a word from the call data:

abbreviation *cut-data* :: *variable-env ⇒ w256 ⇒ w256*
where
cut-data v idx ≡
  read-word-from-bytes (*Word.unat idx*) (*venv-data-sent v*)

Looking up a number of bytes from the memory:

fun *cut-memory* :: *w256 ⇒ nat ⇒* (*w256 ⇒ byte*) *⇒ byte list*
where
cut-memory idx 0 memory = [] |
cut-memory idx (*Suc n*) *memory =*
  memory idx # cut-memory (*idx + 1*) *n memory*

declare *cut-memory.simps* [*simp*]

CALL instruction results in *ContractCall* action when there are enough stack elements (and gas, when we introduce the gas accounting).

definition *call* :: *variable-env ⇒ constant-env ⇒ instruction-result*
where
call v c =
  (*case venv-stack v of*

```
    e0 # e1 # e2 # e3 # e4 # e5 # e6 # rest ⇒
    (if venv-balance v (cenv-this c) < e2 then
       instruction-failure-result v
     else
       InstructionToWorld (ContractCall
         (⦇ callarg-gas = e0
          , callarg-code = Word.ucast e1
          , callarg-recipient = Word.ucast e1
          , callarg-value = e2
          , callarg-data = cut-memory e3 (Word.unat e4) (venv-memory v)
          , callarg-output-begin = e5
          , callarg-output-size = e6 ⦈),
        venv-storage v,
        update-balance (cenv-this c)
          (λ orig ⇒ orig − e2) (venv-balance v),
        Some (∗ saving the variable environment for timing ∗)
          ((venv-advance-pc c v)
           ⦇ venv-stack := rest
           , venv-balance :=
                 update-balance (cenv-this c)
                   (λ orig ⇒ orig − e2) (venv-balance v)
           , venv-memory-usage :=
               M (M (venv-memory-usage v) e3 e4) e5 e6 ⦈, uint e5, uint e6)))
  | - ⇒ instruction-failure-result v)
```

declare *call-def* [*simp*]

DELEGATECALL is slightly different.

```
definition delegatecall :: variable-env ⇒ constant-env ⇒ instruction-result
where
delegatecall v c =
  (case venv-stack v of
    e0 # e1 # e3 # e4 # e5 # e6 # rest ⇒
    (if venv-balance v (cenv-this c) < venv-value-sent v then
       instruction-failure-result v
     else
       InstructionToWorld
         (ContractCall
           (⦇ callarg-gas = e0,
              callarg-code = Word.ucast e1,
              callarg-recipient = cenv-this c,
              callarg-value = venv-value-sent v,
              callarg-data =
                cut-memory e3 (Word.unat e4) (venv-memory v),
              callarg-output-begin = e5,
              callarg-output-size = e6 ⦈),
          venv-storage v, venv-balance v,
          Some (∗ save the variable environment for returns ∗)
            ((venv-advance-pc c v)
```

```
        (| venv-stack := rest
        , venv-memory-usage :=
            M (M (venv-memory-usage v) e3 e4) e5 e6 |), uint e5, uint e6 )))
  | - ⇒ instruction-failure-result v)
```

declare *delegatecall-def* [*simp*]

CALLCODE is another variant.

```
abbreviation callcode :: variable-env ⇒ constant-env ⇒ instruction-result
where
callcode v c ≡
  (case venv-stack v of
    e0 # e1 # e2 # e3 # e4 # e5 # e6 # rest ⇒
    (if venv-balance v (cenv-this c) < e2 then
       instruction-failure-result v
     else
       InstructionToWorld
         (ContractCall
           ((| callarg-gas = e0,
               callarg-code = ucast e1,
               callarg-recipient = cenv-this c,
               callarg-value = e2,
               callarg-data =
                 cut-memory e3 (unat e4) (venv-memory v),
               callarg-output-begin = e5,
               callarg-output-size = e6 |)),
          venv-storage v,
          update-balance (cenv-this c)
            (λ orig ⇒ orig − e2) (venv-balance v),
          Some (∗ saving the variable environment ∗)
            ((venv-advance-pc c v)
              (| venv-stack := rest
              , venv-memory-usage :=
                  M (M (venv-memory-usage v) e3 e4) e5 e6
              , venv-balance :=
                  update-balance (cenv-this c)
                    (λ orig ⇒ orig − e2) (venv-balance v) |), uint e5, uint e6
            )))
  | - ⇒ instruction-failure-result v)
```

CREATE is also similar because the instruction causes execution on another account.

```
abbreviation create ::
  variable-env ⇒ constant-env ⇒ instruction-result
where
create v c ≡
  (case venv-stack v of
    val # code-start # code-len # rest ⇒
      (if venv-balance v (cenv-this c) < val then
```

```
        instruction-failure-result v
      else
        let code =
           cut-memory code-start
             (unat code-len) (venv-memory v) in
        let new-balance =
           update-balance (cenv-this c)
             (λ orig. orig − val) (venv-balance v) in
        InstructionToWorld
          (ContractCreate
            ((| createarg-value = val
             , createarg-code = code |)),
           venv-storage v,
           update-balance (cenv-this c)
             (λ orig. orig − val) (venv-balance v),
           Some (* save the variable environment for returns *)
             ((venv-advance-pc c v)
              (| venv-stack := rest
              , venv-balance :=
                  update-balance (cenv-this c)
                    (λ orig. orig − val) (venv-balance v)
              , venv-memory-usage :=
                  M (venv-memory-usage v) code-start code-len |), 0, 0)))
  | - ⇒ instruction-failure-result v)
```

For implementing RETURN, I need to cut a region from the memory according to the stack elements:

definition

```
venv-returned-bytes v =
  (case venv-stack v of
    e0 # e1 # - ⇒ cut-memory e0 (Word.unat e1) (venv-memory v)
  | - ⇒ [])
```

RETURN is modeled like this:

abbreviation *ret* :: *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where

```
ret v c ≡
   (case venv-stack v of
      e0 # e1 # rest ⇒
         let new-v = v(| venv-memory-usage := M (venv-memory-usage v) e0 e1 |)
in
        InstructionToWorld ((ContractReturn (venv-returned-bytes new-v)),
                           venv-storage v, venv-balance v,
                         None (* No possibility of ever returning to this invocation. *))
   | - ⇒ instruction-failure-result v)
```

STOP is simpler than RETURN:

abbreviation *stop* ::
variable-env ⇒ *constant-env* ⇒ *instruction-result*

where
stop v c ≡
  InstructionToWorld (*ContractReturn* [], *venv-storage v*, *venv-balance v*, *None*)

POP removes the topmost element of the stack:

abbreviation *pop* ::
variable-env ⇒ *constant-env* ⇒ *instruction-result*
where
pop v c ≡ *InstructionContinue* (*venv-advance-pc c*
  v(|*venv-stack* := *tl* (*venv-stack v*)|))

The DUP instructions:

abbreviation *general-dup* ::
nat ⇒ *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
general-dup n v c ≡
  (*if n* > *length* (*venv-stack v*) *then instruction-failure-result v else*
  (*let duplicated* = *venv-stack v* ! (*n* − *1*) *in*
  InstructionContinue (*venv-advance-pc c v*(| *venv-stack* := *duplicated* # *venv-stack v* |))))

A utility function for storing a list of bytes in the memory:

fun *store-byte-list-memory* :: *w256* ⇒ *byte list* ⇒ *memory* ⇒ *memory*
where
  store-byte-list-memory - [] *orig* = *orig*
| *store-byte-list-memory pos* (*h* # *t*) *orig* =
    store-byte-list-memory (*pos* + *1*) *t* (*orig*(*pos* := *h*))

declare *store-byte-list-memory.simps* [*simp*]

Using the function above, it is straightforward to store a byte in the memory.

abbreviation *store-word-memory* :: *w256* ⇒ *w256* ⇒ *memory* ⇒ *memory*
where
store-word-memory pos val mem ≡
  store-byte-list-memory pos (*word-rsplit val*) *mem*

MSTORE writes one word to the memory:

abbreviation *mstore* :: *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
mstore v c ==
  (*case venv-stack v of*
    [] ⇒ *instruction-failure-result v*
  | [-] ⇒ *instruction-failure-result v*
  | *pos* # *val* # *rest* ⇒
      let new-memory = *store-word-memory pos val* (*venv-memory v*) *in*
      InstructionContinue (*venv-advance-pc c*
        v(| *venv-stack* := *rest*

, *venv-memory* := *new-memory*
, *venv-memory-usage* := *M* (*venv-memory-usage v*) *pos 32*
⦈))

MLOAD reads one word from the memory:

abbreviation *mload* :: *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
mload v c ==
(*case venv-stack v of*
pos # *rest* ⇒
let value = *word-rcat* (*cut-memory pos 32* (*venv-memory v*)) *in*
InstructionContinue (*venv-advance-pc c*
v ⦇ *venv-stack* := *value* # *rest*
, *venv-memory-usage* := *M* (*venv-memory-usage v*) *pos 32*
⦈)
| - ⇒ *instruction-failure-result v*)

MSTORE8 writes one byte to the memory:

abbreviation *mstore8* :: *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
mstore8 v c ≡
(*case venv-stack v of*
pos # *val* # *rest* ⇒
let new-memory = (*venv-memory v*)(*pos* := *ucast val*) *in*
InstructionContinue (*venv-advance-pc c*
v⦇ *venv-stack* := *rest*
, *venv-memory-usage* := *M* (*venv-memory-usage v*) *pos 8*
, *venv-memory* := *new-memory* ⦈)
| - ⇒ *instruction-failure-result v*)

For CALLDATACOPY, I need to look at the caller's data as memory.

abbreviation *input-as-memory* :: *byte list* ⇒ *memory*
where
input-as-memory lst idx ≡
(*if length lst* ≤ *unat idx then 0 else lst* ! *unat idx*)

CALLDATACOPY:

abbreviation *calldatacopy* :: *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
calldatacopy v c ≡
(*case venv-stack v of*
(*dst-start* :: *w256*) # *src-start* # *len* # *rest* ⇒
let data =
cut-memory src-start (*unat len*) (*input-as-memory* (*venv-data-sent v*)) *in*
let new-memory = *store-byte-list-memory dst-start data* (*venv-memory v*) *in*
InstructionContinue (*venv-advance-pc c*
v⦇ *venv-stack* := *rest*, *venv-memory* := *new-memory*,

venv-memory-usage := *M* (*venv-memory-usage v*) *dst-start len* ⦈))

CODECOPY copies a region of the currently running code to the memory:

abbreviation *codecopy* :: *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
codecopy v c ≡
 (*case venv-stack v of*
  dst-start # *src-start* # *len* # *rest* ⇒
  let data = *cut-memory src-start* (*unat len*)
     (*program-as-memory* (*cenv-program c*)) *in*
  let new-memory = *store-byte-list-memory dst-start data* (*venv-memory v*) *in*
  InstructionContinue (*venv-advance-pc c*
   v⦇ *venv-stack* := *rest*, *venv-memory* := *new-memory*
   , *venv-memory-usage* := *M* (*venv-memory-usage v*) *dst-start len*
   ⦈)
 | - ⇒ *instruction-failure-result v*)

EXTCODECOPY copies a region of the code of an arbitrary account.:

abbreviation *extcodecopy* :: *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
extcodecopy v c ≡
 (*case venv-stack v of*
  addr # *dst-start* # *src-start* # *len* # *rest* ⇒
  let data = *cut-memory src-start* (*unat len*)
     (*program-as-memory*
      (*venv-ext-program v* (*ucast addr*))) *in*
  let new-memory = *store-byte-list-memory dst-start data* (*venv-memory v*) *in*
  InstructionContinue (*venv-advance-pc c*
   v⦇ *venv-stack* := *rest*, *venv-memory* := *new-memory*,
   venv-memory-usage := *M* (*venv-memory-usage v*) *dst-start len*
   ⦈)
 | - ⇒ *instruction-failure-result v*)

PC instruction could be implemented by *stack-0-1-op*:

abbreviation *pc* :: *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
pc v c ≡
 InstructionContinue (*venv-advance-pc c*
  v⦇ *venv-stack* := *word-of-int* (*venv-pc v*) # *venv-stack v* ⦈)

Logging is currently no-op, until some property about event logging is wanted.

definition *log* :: *nat* ⇒ *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
log n v c =
 InstructionContinue (*venv-advance-pc c*
  (*venv-pop-stack* (*Suc* (*Suc n*)) *v*))

declare *log-def* [*simp*]

For SWAP operations, I first define a swap operations on lists.

definition *list-swap* :: *nat* ⇒ *'a list* ⇒ *'a list option*
where
list-swap n lst =
  (*if length lst* < *n* + *1 then None else*
  Some (*concat* [[*lst* ! *n*], *take* (*n* − *1*) (*drop 1 lst*) , [*lst* ! *0*], *drop* (*1* + *n*) *lst*]))

declare *list-swap-def* [*simp*]

For testing, I prove some lemmata:

lemma *list-swap 1* [*0*, *1*] = *Some* [*1*, *0*]
apply(*auto*)
done

lemma *list-swap 2* [*0*, *1*] = *None*
apply(*auto*)
done

lemma *list-swap 2* [*0*, *1*, *2*] = *Some* [*2*, *1*, *0*]
apply(*auto*)
done

lemma *list-swap 3* [*0*, *1*, *2*, *3*] = *Some* [*3*, *1*, *2*, *0*]
apply(*auto*)
done

lemma*list-swap 1* [*0*, *1*, *2*, *3*] = *Some* [*1*, *0*, *2*, *3*]
apply(*auto*)
done

Using this, I can specify the SWAP operations:

definition *swap* :: *nat* ⇒ *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
swap n v c = (∗ *SWAP3 is modeled by swap 3* ∗)
   (*case list-swap n* (*venv-stack v*) *of*
      None ⇒ *instruction-failure-result v*
    | *Some new-stack* ⇒
      InstructionContinue (*venv-advance-pc c v*(| *venv-stack* := *new-stack* |)))

declare *swap-def* [*simp*]

SHA3 instruciton in the EVM is actually reaak 256. In this development, Keccak256 computation is defined in KEC.thy.

definition *sha3* :: *variable-env* ⇒ *constant-env* ⇒ *instruction-result*
where
sha3 v c ≡
  (*case venv-stack v of*
    start # *len* # *rest* ⇒

```
    InstructionContinue (
      venv-advance-pc c v⦇ venv-stack := keccack
                                  (cut-memory start (unat len) (venv-memory v))
                                # rest
                        , venv-memory-usage := M (venv-memory-usage v) start len
                        ⦈)
  | - ⇒ instruction-failure-result v)
```

declare *sha3-def* [*simp*]

The SUICIDE instruction involves value transfer.

```
definition suicide :: variable-env ⇒ constant-env ⇒ instruction-result
where
suicide v c =
  (case venv-stack v of
     dst # - ⇒
       let new-balance = (venv-balance v)(cenv-this c := 0,
         ucast dst := venv-balance v (cenv-this c) + (venv-balance v (ucast dst))) in
       InstructionToWorld (ContractSuicide,venv-storage v, new-balance, None)
   | - ⇒ instruction-failure-result v)
```

declare *suicide-def* [*simp*]

Finally, using the above definitions, I can define a function that operates an instruction on the execution environments.

```
lemma Word.word-rcat [(0x01 :: byte), 0x02] = (0x0102 :: w256)
apply(simp add: word-rcat-def)
apply(simp add: bin-rcat-def)
apply(simp add: bin-cat-def)
done
```

```
fun instruction-sem :: variable-env ⇒ constant-env ⇒ inst ⇒ instruction-result
where
instruction-sem v c (Stack (PUSH-N lst)) =
     stack-0-1-op v c (Word.word-rcat lst)
| instruction-sem v c (Unknown -) = instruction-failure-result v
| instruction-sem v c (Storage SLOAD) = stack-1-1-op v c (venv-storage v)
| instruction-sem v c (Storage SSTORE) = sstore v c
| instruction-sem v c (Pc JUMPI) = jumpi v c
| instruction-sem v c (Pc JUMP) = jump v c
| instruction-sem v c (Pc JUMPDEST) = stack-0-0-op v c
| instruction-sem v c (Info CALLDATASIZE) = stack-0-1-op v c (datasize v)
| instruction-sem v c (Stack CALLDATALOAD) = stack-1-1-op v c (cut-data v)
| instruction-sem v c (Info CALLER) = stack-0-1-op v c
     (Word.ucast (venv-caller v))
| instruction-sem v c (Arith ADD) = stack-2-1-op v c
     (λ a b. a + b)
| instruction-sem v c (Arith SUB) = stack-2-1-op v c
     (λ a b. a − b)
```

```
| instruction-sem v c (Arith ISZERO) = stack-1-1-op v c
    (λ a. if a = 0 then 1 else 0)
| instruction-sem v c (Misc CALL) = call v c
| instruction-sem v c (Misc RETURN) = ret v c
| instruction-sem v c (Misc STOP) = stop v c
| instruction-sem v c (Dup n) = general-dup n v c
| instruction-sem v c (Stack POP) = pop v c
| instruction-sem v c (Info GASLIMIT) = stack-0-1-op v c
    (block-gaslimit (venv-block v))
| instruction-sem v c (Arith inst-GT) = stack-2-1-op v c
    (λ a b. if a > b then 1 else 0)
| instruction-sem v c (Arith inst-EQ) = stack-2-1-op v c
    (λ a b. if a = b then 1 else 0)
| instruction-sem v c (Annotation a) = eval-annotation a v c
| instruction-sem v c (Bits inst-AND) = stack-2-1-op v c (λ a b. a AND b)
| instruction-sem v c (Bits inst-OR) = stack-2-1-op v c (λ a b. a OR b)
| instruction-sem v c (Bits inst-XOR) = stack-2-1-op v c (λ a b. a XOR b)
| instruction-sem v c (Bits inst-NOT) = stack-1-1-op v c (λ a. NOT a)
| instruction-sem v c (Bits BYTE) =
   stack-2-1-op v c (λ position w.
     if position < 32 then
       ucast ((word-rsplit w :: byte list) ! (unat position))
     else 0)
| instruction-sem v c (Sarith SDIV) = stack-2-1-op v c
    (λ n divisor. if divisor = 0 then 0 else
                       word-of-int ((sint n) div (sint divisor)))
| instruction-sem v c (Sarith SMOD) = stack-2-1-op v c
    (λ n divisor. if divisor = 0 then 0 else
                       word-of-int ((sint n) mod (sint divisor)))
| instruction-sem v c (Sarith SGT) = stack-2-1-op v c
    (λ elm0 elm1. if sint elm0 > sint elm1 then 1 else 0)
| instruction-sem v c (Sarith SLT) = stack-2-1-op v c
    (λ elm0 elm1. if sint elm0 < sint elm1 then 1 else 0)
| instruction-sem v c (Sarith SIGNEXTEND) = stack-2-1-op v c
    (λ len orig.
       of-bl (List.map (λ i.
         if i ≤ 256 − 8 ∗ ((uint len) + 1)
         then test-bit orig (nat (256 − 8 ∗ ((uint len) + 1)))
         else test-bit orig (nat i)
       ) (List.upto 0 256)))
| instruction-sem v c (Arith MUL) = stack-2-1-op v c
    (λ a b. a ∗ b)
| instruction-sem v c (Arith DIV) = stack-2-1-op v c
    (λ a divisor. (if divisor = 0 then 0 else a div divisor))
| instruction-sem v c (Arith MOD) = stack-2-1-op v c
    (λ a divisor. (if divisor = 0 then 0 else a mod divisor))
| instruction-sem v c (Arith ADDMOD) = stack-3-1-op v c
    (λ a b divisor.
        (if divisor = 0 then 0 else (a + b) mod divisor))
```

| instruction-sem v c (Arith MULMOD) = stack-3-1-op v c
  (λ a b divisor.
    (if divisor = 0 then 0 else (a ∗ b) mod divisor))
| instruction-sem v c (Arith EXP) = stack-2-1-op v c
  (λ a exponent. word-of-int ((uint a) ^ (unat exponent)))
| instruction-sem v c (Arith inst-LT) = stack-2-1-op v c
  (λ arg0 arg1. if arg0 < arg1 then 1 else 0)
| instruction-sem v c (Arith SHA3) = sha3 v c
| instruction-sem v c (Info ADDRESS) = stack-0-1-op v c
  (ucast (cenv-this c))
| instruction-sem v c (Info BALANCE) = stack-1-1-op v c
  (λ addr. venv-balance v (ucast addr))
| instruction-sem v c (Info ORIGIN) = stack-0-1-op v c
  (ucast (venv-origin v))
| instruction-sem v c (Info CALLVALUE) = stack-0-1-op v c
  (venv-value-sent v)
| instruction-sem v c (Info CODESIZE) = stack-0-1-op v c
  (word-of-int (program-length (cenv-program c)))
| instruction-sem v c (Info GASPRICE) = stack-0-1-op v c
  (block-gasprice (venv-block v))
| instruction-sem v c (Info EXTCODESIZE) = stack-1-1-op v c
  (λ arg. (word-of-int (program-length (venv-ext-program v (ucast arg)))))
| instruction-sem v c (Info BLOCKHASH) =
  stack-1-1-op v c (block-blockhash (venv-block v))
| instruction-sem v c (Info COINBASE) =
  stack-0-1-op v c (ucast (block-coinbase (venv-block v)))
| instruction-sem v c (Info TIMESTAMP) =
  stack-0-1-op v c (block-timestamp (venv-block v))
| instruction-sem v c (Info NUMBER) =
  stack-0-1-op v c (block-number (venv-block v))
| instruction-sem v c (Info DIFFICULTY) =
  stack-0-1-op v c (block-difficulty (venv-block v))
| instruction-sem v c (Memory MLOAD) = mload v c
| instruction-sem v c (Memory MSTORE) = mstore v c
| instruction-sem v c (Memory MSTORE8) = mstore8 v c
| instruction-sem v c (Memory CALLDATACOPY) = calldatacopy v c
| instruction-sem v c (Memory CODECOPY) = codecopy v c
| instruction-sem v c (Memory EXTCODECOPY) = extcodecopy v c
| instruction-sem v c (Pc PC) = pc v c
| instruction-sem v c (Log LOG0) = log 0 v c
| instruction-sem v c (Log LOG1) = log 1 v c
| instruction-sem v c (Log LOG2) = log 2 v c
| instruction-sem v c (Log LOG3) = log 3 v c
| instruction-sem v c (Log LOG4) = log 4 v c
| instruction-sem v c (Swap n) = swap n v c
| instruction-sem v c (Misc CREATE) = create v c
| instruction-sem v c (Misc CALLCODE) = (∗ callcode v c ∗)
  InstructionAnnotationFailure
  — Since I cannot guarantee anything about CALLCODE, I choose immediate

failure.
| *instruction-sem v c (Misc SUICIDE) = suicide v c*
| *instruction-sem v c (Misc DELEGATECALL) = (∗ delegatecall v c ∗)*
InstructionAnnotationFailure
— Since I cannot guarantee anything about DELEGATECALL, I choose immediate failure.
| *instruction-sem v c (Info GAS) = stack-0-1-op v c (gas v)*
| *instruction-sem v c (Memory MSIZE) =*
stack-0-1-op v c (word-of-int (venv-memory-usage v))

4.10 Programs' Answer to the World

Execution of a program is harder than that of instructions. The biggest difficulty is that the length of the execution is arbitrary. In Isabelle/HOL all functions must terminate, so I need to prove the termination of program execution. In priciple, I could have used gas, but I was lazy to model gas at that moment, so I introduced an artificial step counter. When I prove theorems about smart contracts, the theorems are of the form "for any value of the initial step counter, this and that never happen."

datatype *program-result =*
ProgramStepRunOut — the artificial step counter has run out
| *ProgramToWorld*
contract-action × storage × (address => w256)
× (variable-env × int × int) option
— the program stopped execution because an instruction wants to talk to the world for example because the execution returned, failed, or called an account.
| *ProgramInvalid* — an unknown instruction is found. Maybe this should just count as a failing execution
| *ProgramAnnotationFailure* — an annotation turned out to be false. This does not happen in reality, but this case exists for the sake of the verification.
| *ProgramInit call-env* — This clause does not denote results of program execution. This denotes a state of the program that expects a particular call. This artificial state is used to specify that the incoming call does not overflow the balance of the account. Probably there is a cleaner approach.

Since our program struct contains a list of annotations for each program position, I have a function that checks all annotations at a particular program position:

abbreviation *check-annotations :: variable-env ⇒ constant-env ⇒ bool*
where
check-annotations v c ≡
(let annots = program-annotation (cenv-program c) (venv-pc v) in
List.list-all (λ annot. annot (build-aenv v c)) annots)

The program execution takes two counters. One counter is decremented for each instruction. The other counter is decremented when a backward-jump happens. This setup allows an easy termination proof. Also, during the proofs, I can do case analysis on the number of backwad jumps rather than the number of instructions.

function (*sequential*) *program-sem* :: *variable-env* ⇒ *constant-env* ⇒ *int* ⇒ *nat* ⇒ *program-result*
and *blocked-program-sem* :: *variable-env* ⇒ *constant-env* ⇒ *int* ⇒ *nat* ⇒ *bool* ⇒ *program-result*
where

```
  program-sem - - - 0 = ProgramStepRunOut
| program-sem v c tiny-step (Suc remaining-steps) =
   (if tiny-step ≤ 0 then
     ProgramToWorld(ContractFail,
       venv-storage-at-call v,
       venv-balance-at-call v, None) else
   (if ¬ check-annotations v c then ProgramAnnotationFailure else
   (case venv-next-instruction v c of
      None ⇒ ProgramStepRunOut
    | Some i ⇒
        (case instruction-sem v c i of
          InstructionContinue new-v ⇒
          (strict-if (venv-pc new-v > venv-pc v)
             (blocked-program-sem new-v c
                (tiny-step − 1) (Suc remaining-steps))
             (blocked-program-sem new-v c
                (program-length (cenv-program c)) remaining-steps))
        | InstructionToWorld (a, st, bal, opt-pushed-v) ⇒
          ProgramToWorld (a, st, bal, opt-pushed-v)
        | InstructionAnnotationFailure ⇒ ProgramAnnotationFailure))))
| blocked-program-sem v c l p - = program-sem v c l p
```

by *pat-completeness auto*
termination by *lexicographic-order*

declare *program-sem.psimps* [*simp*]

The following lemma is just for controlling the Isabelle/HOL simplifier.

lemma *unblock-program-sem* [*simp*] : *blocked-program-sem v c l p True* = *program-sem v c l p*
apply(*simp add*: *blocked-program-sem.psimps*)
done

definition *program-sem-blocked* :: *variable-env* ⇒ *constant-env* ⇒ *int* ⇒ *nat* ⇒ *bool* ⇒ *program-result*
where
program-sem-blocked v c internal external - = *program-sem v c internal external*

lemma *program-sem-unblock* :
program-sem-blocked v c internal external True = *program-sem v c internal external*
apply(*simp add*: *program-sem-blocked-def*)
done

4.11 Account's State

In the bigger picture, a contract invocation changes accounts' states. An account has a storage, a piece of code and a balance. Since I am interested in account states in the middle of a transaction, I also need to keep track of the ongoing executions of a single account. Also I need to keep track of a flag indicating if the account has already marked for erasure.

record *account-state* =
  account-address :: *address*
  account-storage :: *storage*
  account-code :: *program*
  account-balance :: *w256*
  account-ongoing-calls :: (*variable-env* × *int* × *int*) *list*
  — the variable environments that are executing on this account, but waiting for calls to finish
  account-killed :: *bool*
  — the boolean that indicates the account has executed SUICIDE in this transaction. The flag causes a destruction of the contract at the end of a transaction.

4.12 Environment Construction before EVM Execution

I need to connect the account state and the program execution environments. First I construct program execution environments from an account state.

Given an account state and a call from the world we can judge if a variable environment is possible or not. The block state is arbitrary. This means we verify properties that hold on whatever block numbers and whatever difficulties and so on. The origin of the transaction is also considered arbitrary.

inductive *build-venv-called* :: *account-state* ⇒ *call-env* ⇒ *variable-env* ⇒ *bool*
where
venv-called:
  bal (*account-address* *a*) =
   (∗ *natural increase is taken care of in RelationalSem.thy* ∗)
     account-balance *a* ⟹
   build-venv-called *a* *env*
   (| (∗ *The stack is initialized for every invocation* ∗)
    venv-stack = []

    (∗ *The memory is also initialized for every invocation* ∗)
   , *venv-memory* = *empty-memory*

    (∗ *The memory usage is initialized.* ∗)
   , *venv-memory-usage* = *0*

    (∗ *The storage is taken from the account state* ∗)
   , *venv-storage* = *account-storage* *a*

    (∗ *The program counter is initialized to zero* ∗)

, *venv-pc* = *0*

(∗ *The balance is arbitrary, except that the balance of this account* ∗)
(∗ *is as specified in the account state plus the sent amount.* ∗)
, *venv-balance* = *bal*(*account-address a* := *bal* (*account-address a*) + *callenv-value env*)

(∗ *the caller is specified by the world* ∗)
, *venv-caller* = *callenv-caller env*

(∗ *the sent value is specified by the world* ∗)
, *venv-value-sent* = *callenv-value env*

(∗ *the sent data is specified by the world* ∗)
, *venv-data-sent* = *callenv-data env*

(∗ *the snapshot of the storage is remembered in case of failure* ∗)
, *venv-storage-at-call* = *account-storage a*

(∗ *the snapshot of the balance is remembered in case of failure* ∗)
, *venv-balance-at-call* = *bal*

(∗ *the origin of the transaction is arbitrarily chosen* ∗)
, *venv-origin* = *origin*

(∗ *the codes of the external programs are arbitrary.* ∗)
, *venv-ext-program* = *ext*

(∗ *the block information is chosen arbitrarily.* ∗)
, *venv-block* = *block*
⦈

declare *build-venv-called.simps* [*simp*]

Similarly we can construct the constant environment. Construction of the constant environment is much simpler than that of a variable environment.

abbreviation *build-cenv* :: *account-state* ⇒ *constant-env*
where
build-cenv a ≡
⦇ *cenv-program* = *account-code a*,
cenv-this = *account-address a* ⦈

Next we turn to the case where the world returns back to the account after the account has called an account. In this case, the account should contain one ongoing execution that is waiting for a call to return.

An instruction is "call-like" when it calls an account and waits for it to return.

abbreviation *is-call-like* :: *inst option* ⇒ *bool*
where
$\textit{is-call-like } i \equiv (i = \textit{Some } (\textit{Misc CALL}) \lor i = \textit{Some } (\textit{Misc DELEGATECALL})$
$\quad \lor i = \textit{Some } (\textit{Misc CALLCODE}) \lor i = \textit{Some } (\textit{Misc CREATE}))$

When an account returns to our contract, the variable environment is recovered from the stack of the ongoing calls. However, due to reentrancy, the balance and the storage of our contract might have changed. So the balance and the storage are taken from the account state provided. Moreover, the balance of our contract might increase because some other contracts might have destroyed themselves, transferring value to our contract.

function *put-return-values* :: *memory* ⇒ *byte list* ⇒ *int* ⇒ *int* ⇒ *memory*
where
$\;\; s \le 0 \Longrightarrow \textit{put-return-values orig - - } s = \textit{orig}$
$| \; s > 0 \Longrightarrow \textit{put-return-values orig } [] \textit{ - } s = \textit{orig}$
$| \; s > 0 \Longrightarrow \textit{put-return-values orig } (h \# t) \; b \; s =$
$\qquad \textit{put-return-values } (\textit{orig}(\textit{word-of-int } b := h)) \; t \; (b + 1) \; (s - 1)$
apply(*auto*)
apply(*case-tac* $b \le 0$; *auto?*)
apply(*case-tac aa*; *auto*)
done

When the control flow comes back to an account state in the form of a return from an account, we build a variable environment as follows. The process is not deterministic because the balance of our contract might have arbitrarily increased.

inductive *build-venv-returned* ::
account-state ⇒ *return-result* ⇒ *variable-env* ⇒ *bool*
where
venv-returned:
$\;\; \textit{is-call-like } (\textit{lookup } (\textit{program-content a-code}) \; (\textit{v-pc} - 1)) \Longrightarrow$
$\;\;\; \textit{new-bal} \ge \textit{a-bal} \Longrightarrow$ (∗ *the balance might have increased* ∗)
$\;\;\;$ *build-venv-returned*

(∗ *here is the first argument* ∗)
(| *account-address* = *a-addr* (∗ *all elements are spelled out for performance* ∗)
, *account-storage* = *a-storage*
, *account-code* = *a-code*
, *account-balance* = *a-bal*
, *account-ongoing-calls* =
((| *venv-stack* = *v-stack*
, *venv-memory* = *v-memory*
, *venv-memory-usage* = *v-memory-usage*
, *venv-storage* = *v-storage*
, *venv-pc* = *v-pc*
, *venv-balance* = *v-balance*
, *venv-caller* = *v-caller*
, *venv-value-sent* = *v-value*

```
        , venv-data-sent = v-data
        , venv-storage-at-call = v-init-storage
        , venv-balance-at-call = v-init-balance
        , venv-origin = v-origin
        , venv-ext-program = v-ext-program
        , venv-block = v-block
        |), mem-start, mem-size) # -
    , account-killed = -
    |)

    (* here is the second argument *)
    r

    (* here is the third argument *)
    ((| venv-stack = 1 # v-stack (* 1 is pushed, indicating a return *)
      , venv-memory =
        put-return-values v-memory (return-data r) mem-start mem-size
      , venv-memory-usage = v-memory-usage
      , venv-storage = a-storage
      , venv-pc = v-pc
      , venv-balance = (update-balance a-addr
                         (λ -. new-bal) (return-balance r))
      , venv-caller = v-caller
      , venv-value-sent = v-value
      , venv-data-sent = v-data
      , venv-storage-at-call = v-init-storage
      , venv-balance-at-call = v-init-balance
      , venv-origin = v-origin
      , venv-ext-program = v-ext-program
      , venv-block = v-block |))
```

declare *build-venv-returned.simps* [*simp*]

The situation is much simpler when an ongoing call has failed because anything meanwhile has no effects.

definition *build-venv-failed* :: *account-state* ⇒ *variable-env option*
where

```
build-venv-failed a =
  (case account-ongoing-calls a of
      [] ⇒ None
   | (recovered, -, -) # - ⇒
      (if is-call-like (* check the previous instruction *)
        (lookup (program-content (account-code a))
         (venv-pc recovered − 1)) then
       Some (recovered
         (|venv-stack := 0 (* indicating failure *) # venv-stack recovered|))
       else None))
```

declare *build-venv-failed-def* [*simp*]

4.13 Account State Update after EVM Execution

Of course the other direction exists for constructing an account state after the program executes.

The first definition is about forgetting one ongoing call.

abbreviation *account-state-pop-ongoing-call* :: *account-state* ⇒ *account-state*
where
account-state-pop-ongoing-call orig ≡
  orig⦇ *account-ongoing-calls* := *tl* (*account-ongoing-calls orig*)⦈

Second I define the empty account, which replaces an account that has destroyed itself.

abbreviation *empty-account* :: *address* ⇒ *account-state*
where
empty-account addr ≡
⦇ *account-address* = *addr*
, *account-storage* = *empty-storage*
, *account-code* = *empty-program*
, *account-balance* = *0*
, *account-ongoing-calls* = []
, *account-killed* = *False*
⦈

And after our contract makes a move, the account state is updated as follows.

definition *update-account-state* ::
account-state ⇒ *contract-action* ⇒ *storage* ⇒ (*address* ⇒ *w256*)
⇒ (*variable-env* × *int* × *int*) *option* ⇒ *account-state*
where
update-account-state prev act st bal v-opt ≡
  prev ⦇
    account-storage := *st*,
    account-balance :=
      (*case act of ContractFail* ⇒ *account-balance prev*
          | - ⇒ *bal* (*account-address prev*)),
    account-ongoing-calls :=
      (*case v-opt of None* ⇒ *account-ongoing-calls prev*
          | *Some pushed* ⇒ *pushed* # *account-ongoing-calls prev*),
    account-killed :=
      (*case act of ContractSuicide* ⇒ *True*
          | - ⇒ *account-killed prev*)⦈

The above definition should be expanded automatically only when the last argument is known to be None or Some _.

lemma *update-account-state-None* [*simp*] :
update-account-state prev act st bal None =
  (*prev* ⦇
    account-storage := *st*,

account-balance :=
  (*case act of ContractFail* ⇒ *account-balance prev*
          | - ⇒ *bal* (*account-address prev*)),
account-ongoing-calls := *account-ongoing-calls prev*,
account-killed :=
  (*case act of ContractSuicide* ⇒ *True*
          | - ⇒ *account-killed prev*) ⦈)
apply(*case-tac act*; *simp add*: *update-account-state-def*)
done

lemma *update-account-state-Some* [*simp*] :
update-account-state prev act st bal (*Some pushed*) =
 (*prev* ⦇
  account-storage := *st*,
  account-balance :=
   (*case act of ContractFail* ⇒ *account-balance prev*
          | - ⇒ *bal* (*account-address prev*)),
  account-ongoing-calls := *pushed* # *account-ongoing-calls prev*,
  account-killed :=
   (*case act of ContractSuicide* ⇒ *True*
          | - ⇒ *account-killed prev*)⦈)
apply(*case-tac act*; *simp add*: *update-account-state-def*)
done

4.14 Controlling the Isabelle Simplifier

This subsection contains simplification rules for the Isabelle simplifier. The main purpose is to prevent the AVL tree implementation to compute both the left insertion and the right insertion when actually only one of these happens.

declare *word-rcat-def* [*simp*]
  unat-def [*simp*]
  bin-rcat-def [*simp*]

I do not allow the AVL library to perform updates at arbitrary moments, because that causes exponentially expensive computation (as measured with the number of elements *)

declare *update.simps* [*simp del*]
declare *lookup.simps* [*simp del*]

Instead, I only allow the following operations to happen (from left to right).

lemma *updateL* [*simp*] : *update x y Leaf* = *Node 1 Leaf* (*x*,*y*) *Leaf*
apply(*simp add*: *update.simps*)
done

lemma *updateN-EQ* [*simp*]: *cmp x a* = *EQ* ⟹ *update x y* (*Node h l* (*a*, *b*) *r*) = *Node h l* (*x*, *y*) *r*
apply(*simp add*: *update.simps*)

done

lemma *updateN-GT* [*simp*]: *cmp x a* = *GT* $\Longrightarrow$ *update x y* (*Node h l* (*a*, *b*) *r*) = *balR l* (*a*, *b*) (*update x y r*)
apply(*simp add*: *update.simps*)
done

lemma *updateN-LT* [*simp*]: *cmp x a* = *LT* $\Longrightarrow$ *update x y* (*Node h l* (*a*, *b*) *r*) = *balL* (*update x y l*) (*a*, *b*) *r*
apply(*simp add*: *update.simps*)
done

lemma *lookupN-EQ* [*simp*]: *cmp x a* = *EQ* $\Longrightarrow$ *lookup* (*Node h l* (*a*, *b*) *r*) *x* = *Some b*
apply(*simp add*: *lookup.simps*)
done

lemma *lookupN-GT* [*simp*]: *cmp x a* = *GT* $\Longrightarrow$ *lookup* (*Node h l* (*a*, *b*) *r*) *x* = *lookup r x*
apply(*simp add*: *lookup.simps*)
done

lemma *lookupN-LT* [*simp*]: *cmp x a* = *LT* $\Longrightarrow$ *lookup* (*Node h l* (*a*, *b*) *r*) *x* = *lookup l x*
apply(*simp add*: *lookup.simps*)
done

lemma *lookupL* [*simp*]: *lookup Leaf x* = *None*
apply(*simp add*: *lookup.simps*)
done

lemma *nodeLL* [*simp*] : *node Leaf a Leaf* == *Node 1 Leaf a Leaf*
apply(*simp add*:*node-def*)
done

lemma *nodeLN* [*simp*] : *node Leaf a* (*Node rsize rl rv rr*) == *Node* (*rsize* + *1*) *Leaf a* (*Node rsize rl rv rr*)
apply(*simp add*:*node-def*)
done

lemma *nodeNL* [*simp*] : *node* $\langle$*lsize*, *ll*, *lv*, *lr*$\rangle$ *a* $\langle\rangle$ == *Node* (*lsize* + *1*) (*Node lsize ll lv lr*) *a Leaf*
apply(*simp add*: *node-def*)
done

lemma *nodeNN* [*simp*] : *node* (*Node lsize ll lv lr*) *a* (*Node rsize rl rv rr*) == *Node* (*max lsize rsize* + *1*) (*Node lsize ll lv lr*) *a* (*Node rsize rl rv rr*)
apply(*simp add*: *node-def*)
done

lemma *balL-neq-NL* [*simp*]:
  $lh \neq Suc\ (Suc\ 0) \Longrightarrow$
    balL (Node lh ll b lr) a Leaf = node (Node lh ll b lr) a Leaf
apply(*simp add*: *balL-def*)
done

lemma *balL-neq-Lr* [*simp*]:
  balL Leaf a r = node Leaf a r
apply(*simp add*: *balL-def*)
done

lemma *balL-neq-NN* [*simp*]:
  $lh \neq Suc\ (Suc\ rh) \Longrightarrow$
    balL (Node lh ll lx lr) a (Node rh rl rx rr) = node (Node lh ll lx lr) a (Node rh rl rx rr)
apply(*simp add*: *balL-def*)
done

lemma *balL-eq-heavy-r-rL* [*simp*]:
  $ht\ bl < ch \Longrightarrow$
    balL (Node (Suc (Suc 0)) bl b (Node ch cl c cr)) a Leaf = node (node bl b cl) c (node cr a Leaf)

apply(*simp add*: *balL-def*)
done

lemma *balL-eq-heavy-r-rN* [*simp*]:
  $hl = Suc\ (Suc\ rh) \Longrightarrow$
    $ht\ bl < ch \Longrightarrow$
    balL (Node hl bl b (Node ch cl c cr)) a (Node rh rl rx rr) = node (node bl b cl) c (node cr a (Node rh rl rx rr))

apply(*simp add*: *balL-def*)
done

lemma *balL-eq-heavy-l* [*simp*]:
  $hl = ht\ r + 2 \Longrightarrow$
    $ht\ bl \geq ht\ br \Longrightarrow$
    balL (Node hl bl b br) a r = node bl b (node br a r)
apply(*simp add*: *balL-def*)
done

lemma *balR-neq-xL* [*simp*]:
  balR l a Leaf = node l a Leaf
apply(*simp add*: *balR-def*)
done

lemma *balR-neq-LN* [*simp*]:

$rh \neq Suc\ (Suc\ 0) \Longrightarrow$
$balR\ Leaf\ a\ (Node\ rh\ rl\ rx\ rr) = node\ Leaf\ a\ (Node\ rh\ rl\ rx\ rr)$
apply(*simp add*: *balR-def*)
done

lemma *balR-neq-NN* [*simp*]:
$rh \neq Suc\ (Suc\ lh) \Longrightarrow$
$balR\ (Node\ lh\ ll\ lx\ lr)\ a\ (Node\ rh\ rl\ rx\ rr) = node\ (Node\ lh\ ll\ lx\ lr)\ a\ (Node\ rh\ rl\ rx\ rr)$
apply(*simp add*: *balR-def*)
done

lemma *balR-eq-heavy-l* [*simp*]:
$bh = ht\ l + 2 \Longrightarrow$
$ch > ht\ br \Longrightarrow$
$balR\ l\ a\ (Node\ bh\ (Node\ ch\ cl\ c\ cr)\ b\ br) =$
$node\ (node\ l\ a\ cl)\ c\ (node\ cr\ b\ br)$
apply(*simp add*: *balR-def*)
done

lemma *balR-eq-heavy-r* [*simp*]:
$bh = ht\ l + 2 \Longrightarrow$
$ht\ bl \leq ht\ br \Longrightarrow$
$balR\ l\ a\ (Node\ bh\ bl\ b\ br) = node\ (node\ l\ a\ bl)\ b\ br$
apply(*simp add*: *balR-def*)
done

There is a common pattern for checking a predicate.

lemma *iszero-iszero* [*simp*] :
$((if\ b\ then\ (1 :: 256\ word)\ else\ 0) = 0) = (\neg\ b)$
apply(*auto*)
done

end

5 What can Happen during a Contract Invocation

This section defines a set of sequence of account states that can appeear during an invocation of a countract. The invocation can be a nested reentrancy, but we focus on one invocation. This means we do not look into details of further reentrancy, but just assume that the inner reentrancy keeps the invariant of the contract. Of course we need to prove that the invariant holds for the code, but when we do that we can assume that the inner nested calls keep the invariants (we can say we are doing mathematical induction on the depth of reentrancy)[8].

[8]This poses troubles dealing with DELEGATECALL and CALLCODE instructions. Currently execution of these instructions causes an immediate annotation failure.

theory *RelationalSem*

imports *Main ./ContractSem*

begin

5.1 Some Possible Changes on Our Account State

The account state might change even when the account's code is not executing.

Between blocks, the account might gain balances because somebody mines Eth for the account. Even within a single block, the balance might increase also while other contracts execute because they might destroy themselves and send their balance to our account. When a transaction finishes, if our contract is marked as killed, it is destroyed. The following relation captures these possibilities.

inductive *account-state-natural-change* :: *account-state* ⇒ *account-state* ⇒ *bool*
where
natural: — The balance of this account might increase whenever the code in our contract is not executing. Some other account might destroy itself and give its balance to our account.
old-bal ≤ *new-bal* ⟹
account-state-natural-change
⦇ *account-address* = *addr*
, *account-storage* = *str*
, *account-code* = *code*
, *account-balance* = *old-bal*
, *account-ongoing-calls* = *going*
, *account-killed* = *killed*
⦈
⦇ *account-address* = *addr*
, *account-storage* = *str*
, *account-code* = *code*
, *account-balance* = *new-bal*
, *account-ongoing-calls* = *going*
, *account-killed* = *killed*
⦈
| *cleaned*: — This happens only at the end of a transaction, but we don't know the transaction boundaries. So this can happen at any moment when there are no ongoing calls.
account-state-natural-change
⦇ *account-address* = *addr*
, *account-storage* = *str*
, *account-code* = *code*
, *account-balance* = *old-bal*
, *account-ongoing-calls* = []
, *account-killed* = *True*

⦈
(*empty-account addr*)

declare *account-state-natural-change.simps* [*simp*]

When the execution comes back from an external call, the account state might have changed arbitrarily. Our strategy is to assume that an invariant is kept here; and later prove that the invariant actually holds (that is, for fewer depth of reentrancy). The whole argument can be seen as a mathematical induction over depths of reentrancy, though this idea has not been formalized yet.

inductive *account-state-return-change* ::
(*account-state* ⇒ *bool*) ⇒ *account-state* ⇒ *account-state* ⇒ *bool*
where
account-return:
invariant
⦇ *account-address* = *addr*
, *account-storage* = *new-str*
, *account-code* = *code*
, *account-balance* = *new-bal*
, *account-ongoing-calls* = *ongoing*
, *account-killed* = *killed*
⦈
⟹
account-state-return-change
invariant
⦇ *account-address* = *addr*
, *account-storage* = *old-str*
, *account-code* = *code*
, *account-balance* = *old-bal*
, *account-ongoing-calls* = *ongoing*
, *account-killed* = *killed*
⦈
⦇ *account-address* = *addr*
, *account-storage* = *new-str*
, *account-code* = *code*
, *account-balance* = *new-bal*
, *account-ongoing-calls* = *ongoing*
, *account-killed* = *killed*
⦈

declare *account-state-return-change.simps* [*simp*]

Next we specify which program results might see a return.

fun *returnable-result* :: *program-result* ⇒ *bool*
where
returnable-result ProgramStepRunOut = *False*

| *returnable-result* (*ProgramToWorld* (*ContractCall* -, -, -, -)) = *True*
| *returnable-result* (*ProgramToWorld* (*ContractCreate* -, -, -, -)) = *True*
| *returnable-result* (*ProgramToWorld* (*ContractSuicide*, -, -, -)) = *False*
| *returnable-result* (*ProgramToWorld* (*ContractFail*, -, -, -)) = *False*
— because we are not modeling nested calls here, the effect of the nested calls are modeled in account_state_return_change
| *returnable-result* (*ProgramToWorld* (*ContractReturn* -, -, -, -)) = *False*
| *returnable-result* (*ProgramInit* -) = *False*
| *returnable-result* *ProgramInvalid* = *False*
| *returnable-result* *ProgramAnnotationFailure* = *False*

5.2 A Round of the Game

Now we are ready to specify the world's turn.

inductive *world-turn* ::
(*account-state* ⇒ *bool*) (∗ *The invariant of our contract*∗)
⇒ (*account-state* ∗ *program-result*)
  (∗ *the account state before the world's move*
    and the last thing our account did ∗)
⇒ (*account-state* ∗ *variable-env*)
  (∗ *the account state after the world's move*
    and the variable environment from which our contract must start. ∗)
⇒ *bool* (∗ *a boolean indicating if that is a possible world's move.* ∗)
where
world-call: — the world might call our contract. We only consider the initial invocation here because the deeper reentrant invocations are considered as a part of the adversarial world. The deeper reentrant invocations are performed without the world replying to the contract.
  (∗ *If a variable environment is built from the old account state* ∗)
  (∗ *and the call arguments*, ∗)
  build-venv-called old-state callargs next-venv ⟹

  (∗ *the world makes a move, showing the variable environment.* ∗)
  world-turn I (*old-state*, *ProgramInit callargs*) (*old-state*, *next-venv*)
| *world-return*: — the world might return to our contract.
  (∗ *If the account state can be changed during reentrancy*,∗)
  account-state-return-change I account-state-going-out account-state-back ⟹

  (∗ *and a variable environment can be recovered from the changed account state*,∗)
  build-venv-returned account-state-back result new-v ⟹

  (∗ *and the previous move of the contract was a call−like action*, ∗)
  returnable-result program-r ⟹

  (∗ *the world can make a move, telling the contract to continue with* ∗)
  (∗ *the variable environment.* ∗)
  world-turn I (*account-state-going-out*, *program-r*)
      (*account-state-pop-ongoing-call account-state-back*, *new-v*)

| *world-fail*: — the world might fail from an account into our contract.
(* *If a variable environment can be recovered from the previous account state*,*)
build-venv-failed account-state-going-out = *Some new-v* ⟹

(* *and if the previous action from the contract was a call*, *)
returnable-result result = *True* ⟹

(* *the world can make a move*, *telling the contract to continue with* *)
(* *the variable environment*. *)
world-turn I (*account-state-going-out*, *result*)
(*account-state-pop-ongoing-call account-state-going-out*, *new-v*)

As a reply, our contract might make a move, or report an annotation failure.

inductive *contract-turn* ::
(*account-state* * *variable-env*) ⇒ (*account-state* * *program-result*) ⇒ *bool*
where
contract-to-world:
(* *Under a constant environment built from the old account state*, *)
build-cenv old-account = *cenv* ⟹

(* *if the program behaves like this*, *)
program-sem old-venv cenv
(*program-length* (*cenv-program cenv*)) *steps*
= *ProgramToWorld* (*act*, *st*, *bal*, *opt-v*) ⟹

(* *and if the account state is updated from the program's result*, *)
account-state-going-out
= *update-account-state old-account act st bal opt-v* ⟹

(* *the contract makes a move and udates the account state*. *)
contract-turn (*old-account*, *old-venv*)
(*account-state-going-out*, *ProgramToWorld* (*act*, *st*, *bal*, *opt-v*))

| *contract-annotation-failure*:
(* *If a constant environment is built from the old account state*, *)
build-cenv old-account = *cenv* ⟹

(* *and if the contract execution results in an annotation failure*, *)
program-sem old-venv cenv
(*program-length* (*cenv-program cenv*)) *steps* = *ProgramAnnotationFailure* ⟹

(* *the contract makes a move*, *indicating the annotation failure*. *)
contract-turn (*old-account*, *old-venv*) (*old-account*, *ProgramAnnotationFailure*)

When we combine the world's turn and the contract's turn, we get one round. The round is a binary relation over a single set.

inductive *one-round* ::
(*account-state* ⇒ *bool*) ⇒
(*account-state* * *program-result*) ⇒

$(account\text{-}state * program\text{-}result) \Rightarrow bool$
where
round:
$world\text{-}turn\ I\ a\ b \Longrightarrow contract\text{-}turn\ b\ c \Longrightarrow one\text{-}round\ I\ a\ c$

5.3 Repetitions of rounds

So, we can repeat rounds and see where they bring us.

The definition is taken from the book Concrete Semantics, and then modified

inductive *star* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
where
refl: $star\ r\ x\ x$ |
step: $r\ x\ y \Longrightarrow star\ r\ y\ z \Longrightarrow star\ r\ x\ z$

The repetition of rounds is either zero-times or once and a repetition.

lemma *star-case* :
$star\ r\ a\ c \Longrightarrow$
$(a = c \lor (\exists\ b.\ r\ a\ b \land star\ r\ b\ c))$
apply(*induction rule*: *star.induct*; *auto*)
done

The next lemma is purely for convenience. Actually the rounds can go nowhere after this invocation fails.

lemma *no-entry-fail* [*dest*!]:
star (*one-round I*)
(*a*, *ProgramToWorld* (*ContractFail*, *st*, *bal*, *v-opt*))
(*b*, *c*) $\Longrightarrow$ *b* = *a* $\land$ *c* = *ProgramToWorld* (*ContractFail*, *st*, *bal*, *v-opt*)
apply(*drule star-case*; *simp*)
apply(*simp add*: *one-round.simps add*: *world-turn.simps*)
done

Similarly, the rounds can go nowhere after this invocation returns.

lemma *no-entry-return* [*dest*!]:
star (*one-round I*)
(*a*, *ProgramToWorld* (*ContractReturn data*, *st*, *bal*, *v-opt*))
(*b*, *c*) $\Longrightarrow$ *b* = *a* $\land$ *c* = *ProgramToWorld* (*ContractReturn data*, *st*, *bal*, *v-opt*)
apply(*drule star-case*; *simp*)
apply(*simp add*: *one-round.simps add*: *world-turn.simps*)
done

Also similarly, the rounds can go nowhere after this invocation causes our contract to destroy itself.

lemma *no-entry-suicide* [*dest*!]:
star (*one-round I*)
(*a*, *ProgramToWorld* (*ContractSuicide*, *st*, *bal*, *v-opt*))
(*b*, *c*) $\Longrightarrow$ *b* = *a* $\land$ *c* = *ProgramToWorld* (*ContractSuicide*, *st*, *bal*, *v-opt*)
apply(*drule star-case*; *simp*)

apply(*simp add*: *one-round.simps add*: *world-turn.simps*)
done

And then the rounds can go nowhere after an annotation failure.

lemma *no-entry-annotation-failure* [*dest*!]:
star (*one-round I*)
    (*a*, *ProgramAnnotationFailure*)
    (*b*, *c*) $\Longrightarrow$ *b* = *a* $\wedge$ *c* = *ProgramAnnotationFailure*
apply(*drule star-case*; *simp*)
apply(*simp add*: *one-round.simps add*: *world-turn.simps*)
done

5.4 How to State an Invariant

For any invariant *I* over account states, now *star* (*one-round I*) relation shows all possibilities during one invocation[9], assuming that the invariant is kept during external calls[10]. The following template traverses the states and states that the invariant is actually kept after every possible round. Also the template states that no annotation failures happen. When I state something, I will be then obliged to prove the statement. This happens in the next section.

I define the conjunction of the properties requested at the final states. The whole thing is more readable when I inline-expand *no-assertion-failure-post*, but if I do that, the *auto* tactic splits out a subgoal for each conjunct. This doubles the already massive number of subgoals.

definition *no-assertion-failure-post* ::
  (*account-state* $\Rightarrow$ *bool*) $\Rightarrow$ (*account-state* $\times$ *program-result*) $\Rightarrow$ *bool*
where
no-assertion-failure-post I fin =
 (*I* (*fst fin*) $\wedge$ (* *The invariant holds.* *)
  snd fin $\neq$ *ProgramAnnotationFailure*) (* *No annotations have failed.* *)

no-assertion-failure is a template for statements. It takes a single argument *I* for the invariant. The invariant is assumed to hold at the initial state. The initial state is when the contract is called (this can be the first invocation of this contract in this transaction, or a reentrancy). The statement will request us to prove that the invariant also holds whenever the invocation loses the control flow. The invocation loses the control flow when the contract returns or fails from the invocation, and also when it calls an account. When the contract calls an account, the invocation does not finish so I need to verify

[9]More precisely, this transitive closure of rounds guides us through all the possible states when the contract loses the control flow.

[10]This assumption about deeper reentrancy should be considered as an induction hypothesis. There has to be a lemma actually perform such induction.

further final states against the postconditions, after the call finishes. The repetition is captured by the transitive closure *star* (*one-round* I).

We prove the invariant when our contract calls out, and we assume that reentrancy into this contract will keep the invariant. The whole argument can be seen as a mathematical induction over the depth of reentrancy. So we can assume that reentrancy of a fewer depth keeps the invariant. This idea comes from Christian Reitwiessner's treatment of reentrancy in Why ML. I have not justified the idea in Isabelle/HOL.

definition *no-assertion-failure* :: (*account-state* ⇒ *bool*) ⇒ *bool*
where
no-assertion-failure (I :: *account-state* ⇒ *bool*) ≡
  (∀ *addr str code bal ongoing killed callenv*.
    I ⦇ *account-address* = *addr*, *account-storage* = *str*, *account-code* = *code*,
      account-balance = *bal*, *account-ongoing-calls* = *ongoing*,
      account-killed = *killed* ⦈ ⟶
  (∀ *fin*. *star* (*one-round* I) (
    ⦇ *account-address* = *addr*, *account-storage* = *str*, *account-code* = *code*,
      account-balance = *bal*, *account-ongoing-calls* = *ongoing*,
      account-killed = *killed* ⦈
  , *ProgramInit callenv*) *fin* ⟶
  no-assertion-failure-post I fin))

5.5 How to State a Pre-Post Condition Pair

After proving a theorem of the above form, I might be interested in a more specific case (e.g. if the caller is not this particular account, nothing should change). For that purpose, here is another template statement. This contains everything above plus an assumption about the initial call, and a conclusion about the state after the invocation.

I pack everything that I want when the contract fails or returns. This definition reduces the number of goals that I need to prove. Without this definition, the *auto* tactic splits a goal $A \Longrightarrow B \wedge C$ into two subgoals $A \Longrightarrow B$ and $A \Longrightarrow C$. When I do complicated case analysis on A, the number of subgoals grow rapidly. So, I define *packed* to be $B \wedge C$ and prevent the *auto* tactic from noticing that it is a conjunction.

The following snippet says the invariant still holds in the observed final state[11] and the postconditions hold there.

definition *postcondition-pack*
where
postcondition-pack I postcondition fin-observed initial-account initial-call fin
=
  (*I fin-observed* ∧

[11] After the invocation finishes, some miner can credit Eth to the account under verification. The "observed" final state is an arbitrary state after such possible balance increases.

postcondition initial-account initial-call (*fin-observed*, *snd fin*))

The whole template takes an invariant I, a *precondition* and a *postcondition*. The statement is about one invocation of the contract. This invocation can be a reentrancy. The initial state is when the contract is invoked, and the final states[12] are when this invocation makes a call to an account, returns or fails. We further consider natural balance increases[13] and use the "observed final state" to evaluate the post condition. Of course, in between, there might be nesting reentrant invocations, that might alter the storage and the balance of the contract.

At the time of invocation, the invariant and the preconditions are assumed. During reentrant calls (that are deeper than the current invocation), the statement will request us to prove that the invariant holds at any moment when the contract loses the control flow (when the contract returns, fails or calls an account). Also we will be requested to prove that the postcondition holds on these occasions. The contract regains the control flow after a deeper call finishes, and the contract would lose the control flow again. All these requirements are captured by the transitive closure of *one-round* relation.

definition *pre-post-conditions* ::
(*account-state* ⇒ *bool*) ⇒ (*account-state* ⇒ *call-env* ⇒ *bool*) ⇒
(*account-state* ⇒ *call-env* ⇒ (*account-state* × *program-result*) ⇒ *bool*) ⇒ *bool*
where
pre-post-conditions
  (*I* :: *account-state* ⇒ *bool*)
  (*precondition* :: *account-state* ⇒ *call-env*⇒ *bool*)
  (*postcondition* :: *account-state* ⇒ *call-env* ⇒
                  (*account-state* × *program-result*) ⇒ *bool*) ≡

  (∗ *for any initial call and initial account state that satisfy* ∗)
  (∗ *the invariant and the precondition*, ∗)
  (∀ *initial-account initial-call*. *I initial-account* ⟶
    precondition initial-account initial-call ⟶

  (∗ *for any final state that are reachable from these initial conditions*, ∗)
  (∀ *fin*. *star* (*one-round I*) (*initial-account*, *ProgramInit initial-call*) *fin* ⟶

  (∗ *the annotations have not failed* ∗)
  snd fin ≠ *ProgramAnnotationFailure* ∧

  (∗ *and for any observed final state after this final state*, ∗)
  (∀ *fin-observed*. *account-state-natural-change* (*fst fin*) *fin-observed* ⟶

[12] Since I am considering all possible executions, there are multiple final states. Also, even when I concetrate on a single execution, every time the contract calls an account, I have to check the invariants. Otherwise, I have no knowledge about what happens during the following reentrancy.

[13] The balnace of an Ethereum account increases naturally when a contract destroys itself and sends its balance to our account, for instance.

(* *the postcondition and the invariant holds.* *)
postcondition-pack
I postcondition fin-observed initial-account initial-call fin)))

end

6 Verification of the Deed Contract

This section focuses on one particular smart contract called the "Deed" contract. The Deed contract is designed as a simple contract trusted with values. So the first aim of the verification is to show that most accounts cannot control the value.

theory *Deed*

imports *Main ../RelationalSem*

begin

6.1 The Code under Verification

The code under verification comes from these commits:

```
github.com/Arachnid/ens: f3334337083728728da56824a5d0a30a8712b60c
github.com/ethereum/solidity: 2d9109ba453d49547778c39a506b0ed492305c16
```

and is produced with this command.

```
$ solc/solc --bin-runtime HashRegistrarSimplified.sol
```

The hex code looks like this
`60606040523615610 06c5760e060020a6000350463...`

I parsed this hex code in a Ruby parser[14] and obtained the following list of instructions.

definition *deed-insts* :: *inst list*
where *deed-insts* =
Stack (*PUSH-N* [*0x60*]) #
Stack (*PUSH-N* [*0x40*]) #
Memory MSTORE #
Info CALLDATASIZE #
Arith ISZERO #
Stack (*PUSH-N* [*0x00*, *0x6c*]) #
Pc JUMPI #

[14] Available in https://github.com/piraira/eth-isabelle

```
Stack (PUSH-N [0xe0]) #
Stack (PUSH-N [0x02]) #
Arith EXP #
Stack (PUSH-N [0x00]) #
Stack CALLDATALOAD #
Arith DIV #
Stack (PUSH-N [0x05, 0xb3, 0x44, 0x10]) #
Dup 2 #
Arith inst-EQ #
Stack (PUSH-N [0x00, 0x6e]) #
Pc JUMPI #
Dup 1 #
Stack (PUSH-N [0x0b, 0x5a, 0xb3, 0xd5]) #
Arith inst-EQ #
Stack (PUSH-N [0x00, 0x7c]) #
Pc JUMPI #
Dup 1 #
Stack (PUSH-N [0x13, 0xaf, 0x40, 0x35]) #
Arith inst-EQ #
Stack (PUSH-N [0x00, 0x89]) #
Pc JUMPI #
Dup 1 #
Stack (PUSH-N [0x2b, 0x20, 0xe3, 0x97]) #
Arith inst-EQ #
Stack (PUSH-N [0x00, 0xaf]) #
Pc JUMPI #
Dup 1 #
Stack (PUSH-N [0x8d, 0xa5, 0xcb, 0x5b]) #
Arith inst-EQ #
Stack (PUSH-N [0x00, 0xc6]) #
Pc JUMPI #
Dup 1 #
Stack (PUSH-N [0xbb, 0xe4, 0x27, 0x71]) #
Arith inst-EQ #
Stack (PUSH-N [0x00, 0xdd]) #
Pc JUMPI #
Dup 1 #
Stack (PUSH-N [0xfa, 0xab, 0x9d, 0x39]) #
Arith inst-EQ #
Stack (PUSH-N [0x01, 0x03]) #
Pc JUMPI #
Dup 1 #
Stack (PUSH-N [0xfb, 0x16, 0x69, 0xca]) #
Arith inst-EQ #
Stack (PUSH-N [0x01, 0x29]) #
Pc JUMPI #
Pc JUMPDEST #
Misc STOP #
Pc JUMPDEST #
```

Info CALLVALUE #
Stack (*PUSH-N* [*0x00*, *0x02*]) #
Pc JUMPI #
Stack (*PUSH-N* [*0x01*, *0x4a*]) #
Stack (*PUSH-N* [*0x01*]) #
Storage SLOAD #
Dup 2 #
Pc JUMP #
Pc JUMPDEST #
Info CALLVALUE #
Stack (*PUSH-N* [*0x00*, *0x02*]) #
Pc JUMPI #
Stack (*PUSH-N* [*0x00*, *0x6c*]) #
Stack (*PUSH-N* [*0x01*, *0x89*]) #
Pc JUMP #
Pc JUMPDEST #
Info CALLVALUE #
Stack (*PUSH-N* [*0x00*, *0x02*]) #
Pc JUMPI #
Stack (*PUSH-N* [*0x00*, *0x6c*]) #
Stack (*PUSH-N* [*0x04*]) #
Stack CALLDATALOAD #
Stack (*PUSH-N* [*0x00*]) #
Storage SLOAD #
Info CALLER #
Stack (*PUSH-N* [*0x01*]) #
Stack (*PUSH-N* [*0xa0*]) #
Stack (*PUSH-N* [*0x02*]) #
Arith EXP #
Arith SUB #
Swap 1 #
Dup 2 #
Bits inst-AND #
Swap 2 #
Bits inst-AND #
Arith inst-EQ #
Stack (*PUSH-N* [*0x01*, *0xf8*]) #
Pc JUMPI #
Stack (*PUSH-N* [*0x00*, *0x02*]) #
Pc JUMP #
Pc JUMPDEST #
Info CALLVALUE #
Stack (*PUSH-N* [*0x00*, *0x02*]) #
Pc JUMPI #
Stack (*PUSH-N* [*0x01*, *0xa0*]) #
Stack (*PUSH-N* [*0x00*]) #
Storage SLOAD #
Stack (*PUSH-N* [*0x01*]) #
Stack (*PUSH-N* [*0xa0*]) #

```
Stack (PUSH-N [0x02]) #
Arith EXP #
Arith SUB #
Bits inst-AND #
Dup 2 #
Pc JUMP #
Pc JUMPDEST #
Info CALLVALUE #
Stack (PUSH-N [0x00, 0x02]) #
Pc JUMPI #
Stack (PUSH-N [0x01, 0xa0]) #
Stack (PUSH-N [0x02]) #
Storage SLOAD #
Stack (PUSH-N [0x01]) #
Stack (PUSH-N [0xa0]) #
Stack (PUSH-N [0x02]) #
Arith EXP #
Arith SUB #
Bits inst-AND #
Dup 2 #
Pc JUMP #
Pc JUMPDEST #
Info CALLVALUE #
Stack (PUSH-N [0x00, 0x02]) #
Pc JUMPI #
Stack (PUSH-N [0x00, 0x6c]) #
Stack (PUSH-N [0x04]) #
Stack CALLDATALOAD #
Stack (PUSH-N [0x00]) #
Storage SLOAD #
Info CALLER #
Stack (PUSH-N [0x01]) #
Stack (PUSH-N [0xa0]) #
Stack (PUSH-N [0x02]) #
Arith EXP #
Arith SUB #
Swap 1 #
Dup 2 #
Bits inst-AND #
Swap 2 #
Bits inst-AND #
Arith inst-EQ #
Stack (PUSH-N [0x02, 0x57]) #
Pc JUMPI #
Stack (PUSH-N [0x00, 0x02]) #
Pc JUMP #
Pc JUMPDEST #
Info CALLVALUE #
Stack (PUSH-N [0x00, 0x02]) #
```

Pc JUMPI #
Stack (*PUSH-N* [*0x00*, *0x6c*]) #
Stack (*PUSH-N* [*0x04*]) #
Stack CALLDATALOAD #
Stack (*PUSH-N* [*0x00*]) #
Storage SLOAD #
Info CALLER #
Stack (*PUSH-N* [*0x01*]) #
Stack (*PUSH-N* [*0xa0*]) #
Stack (*PUSH-N* [*0x02*]) #
Arith EXP #
Arith SUB #
Swap 1 #
Dup 2 #
Bits inst-AND #
Swap 2 #
Bits inst-AND #
Arith inst-EQ #
Stack (*PUSH-N* [*0x02*, *0xc7*]) #
Pc JUMPI #
Stack (*PUSH-N* [*0x00*, *0x02*]) #
Pc JUMP #
Pc JUMPDEST #
Stack (*PUSH-N* [*0x00*, *0x6c*]) #
Stack (*PUSH-N* [*0x04*]) #
Stack CALLDATALOAD #
Stack (*PUSH-N* [*0x00*]) #
Storage SLOAD #
Info CALLER #
Stack (*PUSH-N* [*0x01*]) #
Stack (*PUSH-N* [*0xa0*]) #
Stack (*PUSH-N* [*0x02*]) #
Arith EXP #
Arith SUB #
Swap 1 #
Dup 2 #
Bits inst-AND #
Swap 2 #
Bits inst-AND #
Arith inst-EQ #
Stack (*PUSH-N* [*0x02*, *0xe9*]) #
Pc JUMPI #
Stack (*PUSH-N* [*0x00*, *0x02*]) #
Pc JUMP #
Pc JUMPDEST #
Stack (*PUSH-N* [*0x40*]) #
Dup 1 #
Memory MLOAD #
Swap 2 #

Dup 3 #
Memory MSTORE #
Memory MLOAD #
Swap 1 #
Dup 2 #
Swap 1 #
Arith SUB #
Stack (*PUSH-N* [*0x20*]) #
Arith ADD #
Swap 1 #
Misc RETURN #
Pc JUMPDEST #
Stack (*PUSH-N* [*0x40*]) #
Memory MLOAD #
Stack (*PUSH-N* [*0xbb*, *0x2c*, *0xe2*, *0xf5*, *0x18*, *0x03*, *0xbb*, *0xa1*, *0x6b*, *0xc8*, *0x52*, *0x82*, *0xb4*, *0x7d*, *0xee*, *0xea*, *0x9a*, *0x5c*, *0x62*, *0x23*, *0xea*, *0xbe*, *0xa1*, *0x07*, *0x7b*, *0xe6*, *0x96*, *0xb3*, *0xf2*, *0x65*, *0xcf*, *0x13*]) #
Swap 1 #
Stack (*PUSH-N* [*0x00*]) #
Swap 1 #
Log LOG1 #
Stack (*PUSH-N* [*0x02*, *0x54*]) #
Pc JUMPDEST #
Stack (*PUSH-N* [*0x02*]) #
Storage SLOAD #
Stack (*PUSH-N* [*0xa0*]) #
Stack (*PUSH-N* [*0x02*]) #
Arith EXP #
Swap 1 #
Arith DIV #
Stack (*PUSH-N* [*0xff*]) #
Bits inst-AND #
Arith ISZERO #
Stack (*PUSH-N* [*0x01*, *0xbd*]) #
Pc JUMPI #
Stack (*PUSH-N* [*0x00*, *0x02*]) #
Pc JUMP #
Pc JUMPDEST #
Stack (*PUSH-N* [*0x40*]) #
Dup 1 #
Memory MLOAD #
Stack (*PUSH-N* [*0x01*]) #
Stack (*PUSH-N* [*0xa0*]) #
Stack (*PUSH-N* [*0x02*]) #
Arith EXP #
Arith SUB #
Swap 3 #
Swap 1 #
Swap 3 #

Bits inst-AND #
Dup 3 #
Memory MSTORE #
Memory MLOAD #
Swap 1 #
Dup 2 #
Swap 1 #
Arith SUB #
Stack (*PUSH-N* [*0x20*]) #
Arith ADD #
Swap 1 #
Misc RETURN #
Pc JUMPDEST #
Stack (*PUSH-N* [*0x40*]) #
Memory MLOAD #
Stack (*PUSH-N* [*0x02*]) #
Storage SLOAD #
Stack (*PUSH-N* [*0x01*]) #
Stack (*PUSH-N* [*0xa0*]) #
Stack (*PUSH-N* [*0x02*]) #
Arith EXP #
Arith SUB #
Swap 1 #
Dup 2 #
Bits inst-AND #
Swap 2 #
Info ADDRESS #
Swap 1 #
Swap 2 #
Bits inst-AND #
Info BALANCE #
Dup 1 #
Arith ISZERO #
Stack (*PUSH-N* [*0x08*, *0xfc*]) #
Arith MUL #
Swap 2 #
Stack (*PUSH-N* [*0x00*]) #
Dup 2 #
Dup 2 #
Dup 2 #
Dup 6 #
Dup 9 #
Dup 9 #
Misc CALL #
Swap 4 #
Stack POP #
Stack POP #
Stack POP #
Stack POP #

Arith ISZERO #
Stack (PUSH-N [0x01, 0xf3]) #
Pc JUMPI #
Stack (PUSH-N [0xde, 0xad]) #
Misc SUICIDE #
Pc JUMPDEST #
Stack (PUSH-N [0x00, 0x02]) #
Pc JUMP #
Pc JUMPDEST #
Stack (PUSH-N [0x02]) #
Dup 1 #
Storage SLOAD #
Stack (PUSH-N [0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff]) #
Bits inst-NOT #
Bits inst-AND #
Dup 3 #
Bits inst-OR #
Swap 1 #
Storage SSTORE #
Stack (PUSH-N [0x40]) #
Dup 1 #
Memory MLOAD #
Stack (PUSH-N [0x01]) #
Stack (PUSH-N [0xa0]) #
Stack (PUSH-N [0x02]) #
Arith EXP #
Arith SUB #
Dup 4 #
Bits inst-AND #
Dup 2 #
Memory MSTORE #
Swap 1 #
Memory MLOAD #
Stack (PUSH-N [0xa2, 0xea, 0x98, 0x83, 0xa3, 0x21, 0xa3, 0xe9, 0x7b, 0x82, 0x66, 0xc2, 0xb0, 0x78, 0xbf, 0xee, 0xc6, 0xd5, 0x0c, 0x71, 0x1e, 0xd7, 0x1f, 0x87, 0x4a, 0x90, 0xd5, 0x00, 0xae, 0x2e, 0xaf, 0x36]) #
Swap 2 #
Dup 2 #
Swap 1 #
Arith SUB #
Stack (PUSH-N [0x20]) #
Arith ADD #
Swap 1 #
Log LOG1 #
Pc JUMPDEST #
Stack POP #
Pc JUMP #
Pc JUMPDEST #

Stack (*PUSH-N* [*0x02*]) #
Storage SLOAD #
Stack (*PUSH-N* [*0xa0*]) #
Stack (*PUSH-N* [*0x02*]) #
Arith EXP #
Swap 1 #
Arith DIV #
Stack (*PUSH-N* [*0xff*]) #
Bits inst-AND #
Arith ISZERO #
Arith ISZERO #
Stack (*PUSH-N* [*0x02*, *0x6f*]) #
Pc JUMPI #
Stack (*PUSH-N* [*0x00*, *0x02*]) #
Pc JUMP #
Pc JUMPDEST #
Stack (*PUSH-N* [*0x02*]) #
Dup 1 #
Storage SLOAD #
Stack (*PUSH-N* [*0xff*, *0x00*]) #
Bits inst-NOT #
Bits inst-AND #
Swap 1 #
Storage SSTORE #
Stack (*PUSH-N* [*0x40*]) #
Memory MLOAD #
Stack (*PUSH-N* [*0xde*, *0xad*]) #
Swap 1 #
Stack (*PUSH-N* [*0x03*, *0xe8*]) #
Info ADDRESS #
Stack (*PUSH-N* [*0x01*]) #
Stack (*PUSH-N* [*0xa0*]) #
Stack (*PUSH-N* [*0x02*]) #
Arith EXP #
Arith SUB #
Bits inst-AND #
Info BALANCE #
Dup 5 #
Dup 3 #
Arith SUB #
Arith MUL #
Arith DIV #
Dup 1 #
Arith ISZERO #
Stack (*PUSH-N* [*0x08*, *0xfc*]) #
Arith MUL #
Swap 2 #
Stack (*PUSH-N* [*0x00*]) #

Dup 2 #
Dup 2 #
Dup 2 #
Dup 6 #
Dup 9 #
Dup 9 #
Misc CALL #
Swap 4 #
Stack POP #
Stack POP #
Stack POP #
Stack POP #
Arith ISZERO #
Arith ISZERO #
Stack (PUSH-N [0x01, 0x5c]) #
Pc JUMPI #
Stack (PUSH-N [0x00, 0x02]) #
Pc JUMP #
Pc JUMPDEST #
Stack (PUSH-N [0x00]) #
Dup 1 #
Storage SLOAD #
Stack (PUSH-N [0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff]) #
Bits inst-NOT #
Bits inst-AND #
Dup 3 #
Bits inst-OR #
Swap 1 #
Storage SSTORE #
Stack POP #
Pc JUMP #
Pc JUMPDEST #
Stack (PUSH-N [0x02]) #
Storage SLOAD #
Stack (PUSH-N [0xa0]) #
Stack (PUSH-N [0x02]) #
Arith EXP #
Swap 1 #
Arith DIV #
Stack (PUSH-N [0xff]) #
Bits inst-AND #
Arith ISZERO #
Arith ISZERO #
Stack (PUSH-N [0x03, 0x01]) #
Pc JUMPI #
Stack (PUSH-N [0x00, 0x02]) #
Pc JUMP #
Pc JUMPDEST #

Dup 1 #
Info ADDRESS #
Stack (*PUSH-N* [*0x01*]) #
Stack (*PUSH-N* [*0xa0*]) #
Stack (*PUSH-N* [*0x02*]) #
Arith EXP #
Arith SUB #
Bits inst-AND #
Info BALANCE #
Arith inst-LT #
Arith ISZERO #
Stack (*PUSH-N* [*0x03*, *0x18*]) #
Pc JUMPI #
Stack (*PUSH-N* [*0x00*, *0x02*]) #
Pc JUMP #
Pc JUMPDEST #
Stack (*PUSH-N* [*0x02*]) #
Storage SLOAD #
Stack (*PUSH-N* [*0x40*]) #
Memory MLOAD #
Stack (*PUSH-N* [*0x01*]) #
Stack (*PUSH-N* [*0xa0*]) #
Stack (*PUSH-N* [*0x02*]) #
Arith EXP #
Arith SUB #
Swap 2 #
Dup 3 #
Bits inst-AND #
Swap 2 #
Info ADDRESS #
Bits inst-AND #
Info BALANCE #
Dup 4 #
Swap 1 #
Arith SUB #
Dup 1 #
Arith ISZERO #
Stack (*PUSH-N* [*0x08*, *0xfc*]) #
Arith MUL #
Swap 2 #
Stack (*PUSH-N* [*0x00*]) #
Dup 2 #
Dup 2 #
Dup 2 #
Dup 6 #
Dup 9 #
Dup 9 #
Misc CALL #
Swap 4 #

Stack POP #
Stack POP #
Stack POP #
Stack POP #
Arith ISZERO #
Arith ISZERO #
Stack (*PUSH-N* [*0x02*, *0x54*]) #
Pc JUMPI #
Stack (*PUSH-N* [*0x00*, *0x02*]) #
Pc JUMP #
[]

declare *deed-insts-def* [*simp*]

The next definition translates the list of instructions into an AVL tree. This single step takes around 10 minutes. So I will soon need a program that takes a hex code and produces a binary tree literal in Isabelle/HOL.

definition *content-compiled* :: (*int* ∗ *inst*, *nat*) *tree*
where
content-compiled-def [*simplified*] : *content-compiled* == *program-content-of-lst 0 deed-insts*

The program that appears in the statements of the following lemmata is defined here.

definition *deed-program* :: *program*
where
deed-program-def: *deed-program* =
  (| *program-content* = *content-compiled*
  , *program-length* = *int* (*length deed-insts*)
  , *program-annotation* = *program-annotation-of-lst 0 deed-insts*
  |)

6.2 The Invariant

The invariant is simple. The code of the account is either the one defined above or empty. We have to allow the empty case because this contract might destroy itself.

inductive *deed-inv* :: *account-state* ⇒ *bool*
where
  alive: *account-code a* = *deed-program* ⟹ *deed-inv a*
| *dead*: *account-code a* = *empty-program* ⟹ *deed-inv a*

The program length lookup is optimized.

lemma *prolen* [*simp*] : *program-length deed-program* = *500*
apply(*simp add*: *deed-program-def*)
done

The program annotation lookup is also optimized. There are no annotations in the program under verification.

lemma *proanno* [*simp*] : *program-annotation deed-program n* = []
apply(*simp add*: *deed-program-def*)
done

Here, a term called x is defined. x is by definition equal to the binary tree containing the program, and its definition can be expanded automatically during the proofs.

declare *content-compiled-def* [*simp*]

definition x :: (*int* ∗ *inst*, *nat*) *tree*
where *x-def* [*simplified*] :x ≡ *content-compiled*

declare *content-compiled-def* [*simp del*]

declare *deed-program-def* [*simp del*]

Whenever the content of the program is being looked up, the term x appears, allowing further expansion. Otherwise, *program-content deed-program* stays as just two words. This makes sure that the intermediate goals do not contain the big binary tree as a literal.

lemma *pro-content* [*simp*]: *lookup* (*program-content deed-program*) n == *lookup* x n
apply(*simp add*: *deed-program-def add*: *x-def add*: *content-compiled-def*)
done

declare *deed-insts-def* [*simp del*]
declare *bin-cat-def* [*simp*]

lemma *strict-if-split* :
P (*strict-if* b A B) =
(¬ (b ∧ ¬ P (A *True*) ∨ ¬ b ∧ ¬ P (B *True*)))
apply(*case-tac* b; *auto*)
done

declare
    one-round.simps [*simp*]
    world-turn.simps [*simp*]
    contract-turn.simps [*simp*]
    x-def [*simp*]

6.3 Proof that the Invariant is Kept

The following lemma states that, if the account's code is either empty or the Deed contract's code, that is still the case after an invocation.

lemma *deed-inv-deed-program* [*simp*]:

deed-inv
  (| *account-address* = *addr*
  , *account-storage* = *str*
  , *account-code* = *deed-program*
  , *account-balance* = *bal*
  , *account-ongoing-calls* = *ongoing*
  , *account-killed* = *k*
|)
apply(*simp add*: *deed-inv.simps*)
done

lemma *deed-inv-empty* [*simp*]:
deed-inv
  (| *account-address* = *addr*
  , *account-storage* = *str*
  , *account-code* = *empty-program*
  , *account-balance* = *bal*
  , *account-ongoing-calls* = *ongoing*
  , *account-killed* = *k*
|)
apply(*simp add*: *deed-inv.simps*)
done

The following lemma proves that the code of the Deed contract stays the same or becomes empty. It also proves that no annotations fail, but there are no annotations anyway.

lemma *deed-keeps-invariant* :
no-assertion-failure deed-inv
— The proof is a brute-force case analysis. I believe this can be much shorter, but my aim here was to practice the brute-force case analysis.
apply(*simp only*: *no-assertion-failure-def*)
apply(*rule allI*)
apply(*rule allI*)
apply(*rule allI*)
apply(*rule allI*)
apply(*rule allI*)
apply(*rule allI*)
apply(*rule allI*)
apply(*rule impI*)
apply(*rule allI*)
apply(*rule impI*)
apply(*drule star-case*; *auto*)
  apply(*drule deed-inv.cases*; *auto*)
   apply(*simp add*: *no-assertion-failure-post-def*)
  apply(*simp add*: *no-assertion-failure-post-def*)
 apply(*drule deed-inv.cases*; *auto*)
  apply(*case-tac steps*; *auto*)
  apply(*split strict-if-split*; *auto*)
   apply(*split strict-if-split*; *auto*)

apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split if-splits*; *auto?*)
apply(*split if-splits*; *auto?*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*drule star-case*; *auto simp add*: *no-assertion-failure-post-def*)
apply(*case-tac steps*; *auto*)
apply(*case-tac nat*; *auto*)
apply(*case-tac nata*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac nat*; *auto*)
apply(*case-tac nata*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac nat*; *auto*)
apply(*case-tac nata*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*split if-splits*; *auto?*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*drule star-case*; *auto simp add*: *no-assertion-failure-post-def*)
apply(*case-tac steps*; *auto*)
apply(*case-tac nat*; *auto*)
apply(*case-tac nata*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac nat*; *auto*)
apply(*case-tac nata*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac nat*; *auto*)
apply(*case-tac nata*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac nat*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)

apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split if-splits*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*drule star-case*; *auto simp add*: *no-assertion-failure-post-def*)
apply(*case-tac steps*; *auto*)
apply(*case-tac nat*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split if-splits*; *auto*)
apply(*drule star-case*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac nat*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split if-splits*; *auto*)
apply(*drule star-case*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac nat*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split if-splits*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)

apply(*case-tac nat*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split if-splits*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*drule star-case*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*case-tac steps*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*case-tac steps*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*case-tac steps*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*split strict-if-split*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*case-tac steps*; *auto*)
apply(*simp add*: *no-assertion-failure-post-def*)
apply(*drule deed-inv.cases*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split if-splits*; *auto?*)
apply(*split if-splits*; *auto?*)
apply(*split if-splits*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*case-tac nat*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)

```
      apply(split if-splits; auto)
     apply(split strict-if-split; auto)
    apply(split strict-if-split; auto)
   apply(split strict-if-split; auto)
   apply(split strict-if-split; auto)
   apply(case-tac nat; auto)
  apply(split strict-if-split; auto)
  apply(split strict-if-split; auto)
  apply(split if-splits; auto)
 apply(split strict-if-split; auto)
apply(case-tac steps; auto)
done
```

6.4 Proof about the Case when the Caller is Not the Registrar

I prove another property about the Deed contract. The intention is to prevent attacks. It is not straightforward to define what are attacks. In any case I cannot prevent off-chain attacks (such as bribing). Here I prove a property that most of accounts cannot change certain things in the account. They cannot decrease the balance of the account, and they cannot give themselves the authority to do so. In the current case, the only authorized account is the "registrar," which is remembered at the storage index 0 of the Deed account.

In concrete terms that Isabelle/HOL can understand, the claim can be written as follows:
If

- the caller is not equal to the address stored at index 0,
- the 21st least byte in storage index 2 is not zero,
- the sent value does not overflow the account's balance,
- the account is not marked for destruction at the time of invocation,
- and the invariant holds at the time of invocation,

then, after the invocation,

- the invariant is still kept,
- the balance of the acount is not smaller,
- the 21st least byte in storage index 2 is still not zero,
- the registrar of the account is not changed, and

- the account is still not marked for destruction.

I need some arithmetic preparations. The Solidity compiler and the word library of Isabelle/HOL have different ways of casting *address* into *w256* and back. Some propositions are proved so that these differences disappear automatically.

When an address is converted into a 256-bit word, the represented integer does not change.

lemma *address-cast-eq* :
uint (*ucast* (*a* :: *address*) :: *w256*) = *uint a*
apply(*rule uint-up-ucast*)
apply(*simp add*: *is-up*)
done

The size of an address is 160 bits.

lemma *address-size* [*simp*]:
size (*a* :: *address*) = *160*
apply(*simp add*: *word-size*)
done

All addresses are less than 2^{160}.

lemma *address-small′* [*simplified*]:
uint (*a* :: *address*) < *2* ^ *size a*
apply(*simp only*: *uint-range-size*)
done

All addresses cast to words are still small.

lemma *address-small*:
(*ucast* (*a* :: *address*) :: *w256*) < *2* ^ *160*
apply(*simp only*: *word-less-alt*)
apply(*simp add*: *address-cast-eq*)
apply(*rule address-small′*)
done

declare *mask-def* [*simp*]

When you cast an address to word, and take the least 160 bits, that's the same thing as just casting the address.

lemma *address-cast-and* [*simplified*] :
(*mask 160* :: *w256*) *AND ucast* (*a* :: *address*) = (*ucast* (*a* :: *address*) :: *w256*)
apply(*simp only*: *word-bool-alg.conj-commute*)
apply(*rule less-mask-eq*)
apply(*rule address-small*)
done

declare *address-cast-and* [*simp*]

Casting a word to an address can be done after truncating to 160 bits.

lemma *casting-and-truncation*:
word-of-int (*bintrunc 160* (*uint* (*w* :: *w256*))) = (*word-of-int* (*uint w*) :: *address*)
apply(*rule wi-bintr*)
apply(*auto*)
done

When two numbers are equal as words, they are also equal as addresses.

lemma *finer*:
(*word-of-int p* :: *w256*) = *word-of-int q* $\Longrightarrow$
(*word-of-int p* :: *address*) = *word-of-int q*
apply(*simp only*: *word.abs-eq-iff*)
apply(*simp*)
apply(*simp add*: *Bit-Representation.bintrunc-mod2p*)
apply(*simp add*: *Divides.zmod-eq-dvd-iff*)
apply(*rule Rings.comm-monoid-mult-class.dvd-trans*; *auto*)
done

If a word is masked to 160-bits and compared with a casted address, the word is compared against the address.

lemma *addr-case-eq* [*simplified*]:
(*w* :: *w256*) *AND* (*mask 160* :: *w256*) = *ucast*(*a* :: *address*)
$\Longrightarrow$ *ucast w* = *a*
apply(*simp only*: *and-mask-bintr*)
apply(*simp only*: *ucast-def*)
apply(*simp only*: *casting-and-truncation* [*symmetric*])
apply(*drule finer*)
apply(*simp*)
done

declare *addr-case-eq* [*dest*]

declare *mask-def* [*simp del*]

Now we are ready to state and prove the lemma.

lemma *deed-only-registrar-can-spend* :
pre-post-conditions

(∗ *The invariant which is assumed at the beginning of this invocation*,
assumed to be kept during reentrancy calls, *and*
proven at the time of return or failure from this invocation. ∗)
deed-inv

(∗ *The additional conditions that are assumed at the beginning of this invocation*:
∗)
(λ *init-state init-call*.

(∗ *the caller is not the regsitrar*; ∗)

ucast (*account-storage init-state 0*) $\neq$ *callenv-caller init-call*

(∗ *the Deed contract is still marked active*; ∗)
$\wedge$ (*255 AND account-storage init-state 2 div 2 ˆ 160* $\neq$ *0*)

(∗ *the call does not overflow the balance*; ∗)
$\wedge$ *account-balance init-state* + *callenv-value init-call* $\geq$ *account-balance init-state*

(∗ *the account is not marked as destroyed*. ∗)
$\wedge$ $\neg$ (*account-killed init-state*))

(∗ *The additional conditions that are proven to*
∗ *hold when this invocation returns or fails*. ∗)
(λ *init-state* - (*post-state*, -).

(∗ *The balance has not decreased*. ∗)
account-balance init-state $\leq$ *account-balance post-state*

(∗ *The account is still not marked for destruction*
(*i.e. the account has not executed self*−*destruction*). ∗)
$\wedge$ $\neg$ (*account-killed post-state*)

(∗ *The Deed contract is still marked as active*. ∗)
$\wedge$ (*255 AND account-storage post-state 2 div 2 ˆ 160* $\neq$ *0*)

(∗ *The registrar of the contract remains the same*. ∗)
$\wedge$ *account-storage init-state 0* = *account-storage post-state 0*)

— The proof is again a brute-force case analysis.
apply(*simp add*: *pre-post-conditions-def*)
apply(*rule allI*)
apply(*rule impI*)
apply(*drule deed-inv.cases*; *auto*)
apply(*drule star-case*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)

apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*case-tac steps*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*drule star-case*; *auto*)
apply(*simp add*: *postcondition-pack-def add*: *deed-inv.simps*)
apply(*case-tac steps*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*case-tac a*; *simp add*: *postcondition-pack-def add*: *deed-inv.simps*)
apply(*split strict-if-split*; *auto*)
apply(*case-tac a*; *simp add*: *postcondition-pack-def add*: *deed-inv.simps*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*case-tac a*; *simp add*: *postcondition-pack-def add*: *deed-inv.simps*)
apply(*case-tac a*; *simp add*: *postcondition-pack-def add*: *deed-inv.simps*)
apply(*split strict-if-split*; *auto*)
apply(*split strict-if-split*; *auto*)
apply(*case-tac a*; *simp add*: *postcondition-pack-def add*: *deed-inv.simps*)

```
          apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
        apply(split strict-if-split; auto)
          apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
          apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
        apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
      apply(split strict-if-split; auto)
          apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
        apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
      apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
    apply(split strict-if-split; auto)
      apply(split strict-if-split; auto)
      apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
    apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
  apply(split strict-if-split; auto)
    apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
  apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
  apply(split strict-if-split; auto)
    apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
  apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
 apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
 apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
apply(case-tac steps; auto)
apply(split strict-if-split; auto)
apply(split strict-if-split; auto)
 apply(split strict-if-split; auto)
  apply(split strict-if-split; auto)
   apply(split strict-if-split; auto)
    apply(split strict-if-split; auto)
     apply(split strict-if-split; auto)
      apply(split strict-if-split; auto)
       apply(split strict-if-split; auto)
       apply(split strict-if-split; auto)
       apply(split strict-if-split; auto)
      apply(split strict-if-split; auto)
     apply(split strict-if-split; auto)
     apply(split strict-if-split; auto)
    apply(split strict-if-split; auto)
   apply(split strict-if-split; auto)
  apply(split strict-if-split; auto)
  apply(split strict-if-split; auto)
 apply(split strict-if-split; auto)
apply(split strict-if-split; auto)
apply(drule star-case; auto)
apply(case-tac steps; auto)
apply(split strict-if-split; auto)
 apply(split strict-if-split; auto)
  apply(split strict-if-split; auto)
   apply(split strict-if-split; auto)
    apply(split strict-if-split; auto)
```

```
        apply(split strict-if-split; auto)
         apply(split strict-if-split; auto)
          apply(split strict-if-split; auto)
           apply(split strict-if-split; auto)
            apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
           apply(split strict-if-split; auto)
            apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
          apply(split strict-if-split; auto)
           apply(split strict-if-split; auto)
           apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
          apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
         apply(split strict-if-split; auto)
          apply(split strict-if-split; auto)
          apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
         apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
        apply(split strict-if-split; auto)
          apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
         apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
        apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
       apply(split strict-if-split; auto)
      apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
        apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
       apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
      apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
     apply(split strict-if-split; auto)
        apply(split strict-if-split; auto)
       apply(split strict-if-split; auto)
      apply(split strict-if-split; auto)
     apply(split strict-if-split; auto)
      apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
     apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
    apply(split strict-if-split; auto)
     apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
    apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
    apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
   apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
  apply(drule star-case; auto)
   apply(case-tac steps; auto)
  apply(case-tac steps; auto)
 apply(drule star-case; auto)
   apply(simp add: postcondition-pack-def add: deed-inv.simps)
  apply(case-tac steps; auto)
  apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
 apply(case-tac steps; auto)
apply(drule star-case; auto)
 apply(case-tac steps; auto)
apply(case-tac a; simp add: postcondition-pack-def add: deed-inv.simps)
done
```

It takes 45 minutes to compile this proof on my machine. Ten minutes are spent translating the list of instructions into an AVL tree. Most of the rest is spent on following the proofs of the last two lemmata.

end