A simple model of smart contracts in Agda

Fahad F. Alhabardi[1] and Anton Setzer[2]

[1,2]Swansea University, Dept. of Computer Science
[1]fahadalhabardi@gmail.com
[2]a.g.setzer@swansea.ac.uk http://www.cs.swan.ac.uk/~csetzer/

Abstract

This paper defines a simple basic model of smart contracts in Agda. In the previous work [1], we verified smart contracts in Bitcoin. The aim of this paper is the first step towards transferring this work to the Solidity-style [8] smart contracts of Ethereum, namely, to develop a model. This model is much more complex than that used for Bitcoin because contracts in Ethereum are object-oriented. We build a simple model which supports simple execution, calling of other contracts and functions and which refers to addresses and messages.

Smart contracts are one of the main applications of Blockchain. In Blockchain, smart contracts are programs that automatically run when certain predetermined criteria are satisfied [9, 5]. The simplest example of a smart contract is used to buy and sell goods or services: The purchaser deposits funds on the Blockchain for the seller. The funds are not released to the seller until a second signature is obtained from the purchaser upon receipt of the items. If the products are not delivered on time, the customer is refunded [7]. Simple smart contracts like the above can be written in Bitcoin in Script [4]. The cryptocurrency Ethereum has a much more powerful Turing complete machine language, the Ethereum Virtual Machine (EVM), which allows calls to other contracts. There are two high-level languages which compile into the EVM, Solidity and Vyper [2, 3]. Other cryptocurrencies have their own languages [2].

As smart contract failures can trigger huge financial losses, accuracy and security are compulsory in smart contracts before deploying on the Blockchain network because once deployed, the contract is immutable. [10]. Verification of smart contracts is costly, yet it is invaluable in helping to limit the financial implications of poorly designed contracts. An example of poor design is evident from hacking the Distributed Autonomous Organization (DAO) smart contract in 2016 [6, 7]. DAO is a contract issued on the cryptocurrency Ethereum and is an investor-directed venture capital fund based on smart contracts. A flaw in the smart contract code of DAO was exploited by cyber criminals when the fund's market value reached US$ 150 million.

In this paper, we build as a first step towards the verification of smart contracts a model of smart contracts in the theorem prover Agda. In the Ethereum virtual machine, one can call functions using arguments which serialise the data passed on to them. In our model, we abstract from this by defining a data type of messages. Messages are natural numbers, or lists of messages. This allows to represent elements of data types as messages; for instance, an array can be represented as a list of messages, and a map can be represented as a list of pairs consisting of a key and the element it is mapped to (both represented as messages).

```
data Msg : Set where nat : (n : ℕ) → Msg
                     list : (l : List Msg) → Msg
```

After that, we define smart contracts mutually recursively as a coinductive record SmartContractExecStep, which allows conditionals, sequential compositions, and loops, together with the data type SmartContractExec as follows:

```
record SmartContractExecStep : Set where
  coinductive
```

```
    field  calledAddress  : Address
           calledFunction : FunctionName
           calledMsg      : Msg
           cont           : Msg → SmartContractExec
data SmartContractExec : Set where
  return : Msg → SmartContractExec
  call   : SmartContractExecStep → SmartContractExec
  error  : ErrorMsg → SmartContractExec
```

Our smart contracts execution step consists of the address to call (calledAddress), the function name to call (calledFunction), followed by the message call (calledMsg), and the continuation (cont) which determines the next execution step depending on the message returned when the call to the function has finished. SmartContractExec determines the three steps how to continue in the execution: return, which will terminate execution and return its argument, call which will make a call SmartContractExecStep, and then continue as defined by its continuation argument, and error, which will return an error.

The Ledger is a function which depending on addresses, function names, and messages (which are the arguments to the function) returns a SmartContractExecStep:

```
Ledger = Address → FunctionName → Msg → SmartContractExec
```

In order to compute the execution of a call to a smart contract, we define a smart contract stack (ExecutionStack), each element of which determines depending on the result of the current execution the next SmartContractExecStep:

```
ExecutionStack = List (Msg → SmartContractExec)
```

The state of executing consists of the execution stack and the current code to be executed:

```
record StateExecFun : Set where
  constructor stateEF
  field executionStack : ExecutionStack
        nextstep       : SmartContractExec
```

We define stepEF, the one step execution of a smart contract, and stepEFntimes, which iterates it n times, corresponding to execution with a simple form of gas limit:

```
stepEF       : Ledger → StateExecFun → StateExecFun
stepEFntimes : Ledger → StateExecFun → ℕ → StateExecFun
```

As an example, we build a ledger which has at address 0 a function `"f1"` which calls contract with address 1, function `"g1"`, message (nat n) and will terminate with the result returned. Furthermore it has at address 1 a function `"g1"` which just increments a natural number argument by 1. For all other addresses, functions, and arguments, it is undefined:

```
testLedger : Ledger
testLedger 0 "f1" (nat n) = call (smartContractExecStep 1 "g1" (nat n) return)
testLedger 1 "g1" (nat n) = return (nat (suc n))
testLedger ow ow' ow''    = error (strErr " Error undefined")
```

To conclude, we have built a basic smart contract model that supports execution. In the next step, we will add state, gas cost and amount of money. Furthermore, we will include more complex operations, such as transactions, in our model and deal with interactive programs in Agda. Moreover, we will verify smart contracts in our model by using the weakest preconditions, extending the work in [1]. Weakest preconditions can be used to determine for a smart contract the conditions required to carry out a certain transfer.

References

[1] Fahad F. Alhabardi, Arnold Beckmann, Bogdan Lazar, and Anton Setzer. Verification of Bitcoin Script in Agda Using Weakest Preconditions for Access Control. In *27th International Conference on Types for Proofs and Programs (TYPES 2021)*, volume 239 of *LIPIcs*, pages 1:1–1:25, Dagstuhl, Germany, 2022. Leibniz-Zentrum für Informatik. doi: https://doi.org/10.4230/LIPIcs.TYPES.2021.1.

[2] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. Verification of smart contracts: A survey. *Pervasive and Mobile Computing*, 67:1–19, 2020. doi:10.1016/j.pmcj.2020.101227.

[3] Massimo Bartoletti and Roberto Zunino. BitML: A Calculus for Bitcoin Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 83–100, New York, NY, USA, 2018. Association for Computing Machinery. doi:http://dx.doi.org/10.1145/3243734.3243795.

[4] Harris Brakmić. Bitcoin Script. In *Bitcoin and Lightning Network on Raspberry Pi: Running Nodes on Pi3, Pi4 and Pi Zero*, pages 201–224, Berkeley, CA, 2019. Apress. doi:10.1007/978-1-4842-5522-3_7.

[5] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978309.

[6] Zeinab Nehaï, Pierre-Yves Piriou, and Frédéric Daumas. Model-Checking of Smart Contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 980–987, 2018. doi:http://dx.doi.org/10.1109/Cybermatics_2018.2018.00185.

[7] Anton Setzer. Modelling Bitcoin in Agda. *CoRR*, abs/1804.06398, 2018. URL: http://arxiv.org/abs/1804.06398, arXiv:1804.06398.

[8] Solidity Community. Solidity documentation, Retrieved 10 March 2023. Availabe from https://docs.soliditylang.org/en/v0.8.19/.

[9] Nick Szabo. Smart contracts: Building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought*, 18(2), 1996. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[10] Maximilian Wohrer and Uwe Zdun. Smart Contracts: Security patterns in the Ethereum ecosystem and Solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8, 2018. doi:10.1109/IWBOSE.2018.8327565.

Rank-polymorphic arrays within dependent types

Artjoms Šinkarovs[1]*and Sven-Bodo Scholz[2]

[1] Heriot-Watt University
a.sinkarovs@hw.ac.uk
[2] Radboud University
svenbodo.scholz@ru.nl

Many array languages such as APL [6], J [10], Futhark [4], or SaC [9] cater for multi-dimensional arrays as first class citizens. These languages have two main advantages. Firstly, they give rise to concise specifications of numerical algorithms that use array combinators rather than explicit indexing as often found in imperative languages such as Fortran or C. Secondly, as arrays have a very regular structure, many computations on arrays can be automatically parallelised, which leads to efficient executions on a range of parallel platforms [2, 3, 8, 5].

Rank polymorphism is the ability of functions to be applied to arrays of arbitrary ranks. Rank polymorphism is important for two reasons. Firstly, it gives rise to more general array combinators such as map, fold, take, transpose, *etc.* Secondly, the structure of the nesting can be used to enforce non-trivial traversals through sub-arrays which is often the basis for advanced parallel algorithms such as scan or blocked matrix multiply.

In this talk we present how rank-polymorphic arrays can be embedded within a dependently-typed language. On the one hand, our embedding offers the generality of the specifications found in array languages. On the other hand, we guarantee safe indexing and offer a way to reason about concurrency patterns within the given algorithm.

We present the key ingredients of the array framework in Agda. We start with the definition of an array theory.

```
record Array : Set₁ where
  field
    S : Set
    P : S → Set
    ι : ℕ → S
    _⊗_ : S → S → S
    ι-↔ : ∀ {n} → P (ι n) ↔ Fin n
    ⊗-↔ : ∀ {s p} → P (s ⊗ p) ↔ (P s × P p)
    Ar : S → Set → Set
    Ar-↔ : ∀ {s X} → (P s → X) ↔ Ar s X
```

Array shapes S are binary trees with natural numbers as leaves. Array indices P are indexed by shapes, representing trees of natural numbers of the same shape as the index, but where all leaves are component-wise smaller than the shape components. For example, for some shape $(\iota\ a \otimes (\iota\ b \otimes \iota\ c))$ the index is of the form $(i,(j,k))$ where $i < a$, $j < b$ and $k < c$. Array theory does not insist on a particular implementation of S and P but it requires the chosen implementation to be isomorphic to such trees (ι-↔ and ⊗-↔). Finally, arrays (Ar) are indexed by the shape and the element type, and we ask that arrays are representable functors (Ar-↔).

By expanding isomorphisms in the array theory, we get a number of useful array combinators as model constructions. For some (A : Array), we have:

*This work is supported by the Engineering and Physical Sciences Research Council through the grant EP/N028201/1.