Proving Conditional Termination for Smart Contracts

Ton Chanh Le
Stevens Institute of Technology

Lei Xu, Lin Chen, Weidong Shi
University of Houston

ABSTRACT

Termination of smart contracts is crucial for any blockchain system's security and consistency, especially for those supporting Turing-complete smart contract languages. *Resource-constrained* blockchain systems, like Ethereum and Hyperledger Fabric, could prevent smart contracts from terminating properly when the pre-allocated resources are not sufficient. The Zen system utilizes the dependent type system of the programming language F* to prove the termination of smart contracts for all inputs during compilation time. Since the smart contract execution usually depends on the current blockchain state and user inputs, this approach is not always successful. In this work, we propose a lazy approach by statically proving *conditional termination* and *non-termination* of a smart contract to determine input conditions under which the contract terminates or not. Prior to the execution of the smart contract, the *proof-carrying blockchain system* will check that its current state and the contract's input satisfy the termination conditions in order to determine if the contract is qualified (i.e., eventually terminating) to run on the chain.

CCS CONCEPTS

• **Software and its engineering → Software verification**; **Automated static analysis**;

KEYWORDS

blockchain, smart contracts, termination, non-termination

ACM Reference Format:
Ton Chanh Le and Lei Xu, Lin Chen, Weidong Shi. 2018. Proving Conditional Termination for Smart Contracts. In *BCC'18: The 2nd ACM Workshop on Blockchains, Cryptocurrencies and Contracts, June 4, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3205230.3205239

1 INTRODUCTION

Smart contracts, first defined by Szabo [11], are computer programs that encode business logics and rules to process transactions in a blockchain system. Beside functional correctness, termination of smart contracts is also crucial for any blockchain system's security and consistency. Unlike other types of computer programs in which non-termination may be an expected behavior (e.g., reactive systems), non-termination of smart contracts should be considered as *liveness bugs* since accidentally allowing a non-terminating smart contract to execute on the blockchain will definitely make the whole system collapse. This non-terminating smart contract will run on every node in the decentralized system forever. Any individual attempt to terminate its computation will make the network consensus broken since nodes cannot agree on the final outcome of these interrupted computations. The Bitcoin system [9] avoids this problem by allowing only a *Turing-incomplete* language (without supporting loops and recursion) for their scripts. However, loops and recursion are unavoidable if you want to construct an ecosystem supporting more complex applications and rich features; for example, to develop an application with recurring payment.

Currently, the blockchain platform Ethereum [1] relies on a transaction fee system (known as *gas*) to prevent smart contracts run forever on its *Turing-complete* virtual machine with financial penalties. When a user invokes an on-chain smart contract, he/she has to specify an amount of gas to pay for the contract's execution. If the pre-specified gas runs out before the contract's normal termination, the execution will be interrupted by an exception and its computation will be reverted to the initial state. On the other hand, Hyperledger Fabric [2] and other private blockchain systems, which do not offer cyptocurrencies in their ecosystems, use the timer to constrain the execution time of smart contracts. We call these systems as *resource-constrained* blockchain systems. The approaches of these systems could prevent a smart contract from terminating properly when the pre-allocated resource (either gas or time) is not sufficient. With the presence of loops and recursive calls, predicting an accurate amount of resource needed for the contract's execution is not an easy task.

The Zen system [4] proposes another approach to utilize F* [10] as the programming language of their smart contracts. F* is a functional programming language with a strong type system designed for program formal verification. The type system of F* is expressive enough to precisely and concisely specify various security, functional correctness properties, as well as termination of programs. The F* type checker proves that the programs meet their specifications during compilation time. Therefore, if a smart contract in F* is well-typed, it can be safely deloyed and executed on the blockchain system. However, the F* system can only automatically infer ranking functions for proving the termination of simple programs. For complex programs, like the Ackermann function[1], the system requires users to manually provide valid ranking functions to support the termination proofs. In addition, since the smart contract execution usually depends on the current blockchain state and user inputs, the users have to also determine the preconditions in which the contracts terminate. Again, these both tasks are not straightforward.

To address these shortcomings in proving termination of smart contracts, we propose a lazy approach by statically proving their *conditional termination* and *non-termination*. In this approach, when a smart contract is submitted to the blockchain, the system first

[1] https://en.wikipedia.org/wiki/Ackermann_function

BCC'18, June 4, 2018, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5758-6/18/06…$15.00
https://doi.org/10.1145/3205230.3205239

automatically determines preconditions of the contract's inputs and the chain states under which its execution terminates or not. These conditions, in the form of logical formulas, are verifiable and then recorded into the blockchain as a metadata of the contract. Later, when the contract is invoked by a transaction, the system will check if the current chain state and the contract's input satisfy any recorded termination precondition. If they are, then the contract are safely executed since under this condition, we can assure that the contract's execution will eventually terminate in a finite time.

Related work. There have been several approaches to prove conditional termination and non-termination of imperative programs. The problem was first addressed by Cook et al. [3], which determines termination preconditions of a program from bounded, but not decreasing, potential ranking functions. Later, Le et al. [7, 8] develop HipTNT+, a modular termination and non-termination analyzer, by utilizing abductive reasoning and case analysis to incrementally construct a complete summary of both terminating and non-terminating behaviors of each method in a program. Moreover, the outcomes of HipTNT+ can be stored in the form of formal specifications of a verification framework [6]. This framework facilitates the validation of an initial state of a program against its termination specification to decide on acceptance of this terminating state or to explain why this state is rejected due to non-termination.

2 BACKGROUND

We briefly introduce the background of blockchain systems, smart contracts, and concepts related to the termination problem such as ranking functions and termination specification language.

2.1 Blockchain and smart contracts

The concept of a blockchain system was first introduced by Nakamoto [9] in 2008. Typically, a blockchain is usually defined as an open, distributed ledger that records transactions between parties. A blockchain is implemented as a decentralized system of a chain of blocks. Each block contains some information, a timestamp, and more importantly, a hash pointer to securely link to its precedence block using cryptography. These hash pointers, together with a consensus protocol on the sequence and content of blocks, ensure the system's consistency and immutability.

Smart contract is an important feature of many blockchain systems. These smart contracts are computer programs stored on a blockchain to process transactions based on embedded business logics and rules. Szabo [11] defined a general concept of smart contracts, as recalled in Definition 2.1.

Definition 2.1 (Smart contract). A smart contract is a set of promises, specified in a digital form, including protocols within which the parties perform on these promises.

2.2 Proving termination and non-termination

A traditional method for proving program termination is to find an appropriate well-founded relation, called ranking function, that maps program states to a well-ordered domain [12]. Since the well-founded relation contains no infinite descending chains of elements, the program execution does not have any infinite state sequences, thus its termination can be concluded. We recall the formal definition of ranking functions as follows:

Definition 2.2 (Ranking function). Given a program $P = (S, R)$ defined as a pair of a set of states S and a transition relation $R \subseteq S \times S$. A ranking function is a mapping $r : S \mapsto \mathbb{A}$ from the set of states S into a well-ordered set $(\mathbb{A}, \preceq)$ such that for each transition $(s, s') \in R$, $r(s) \succ r(s')$ and $r(s) \succeq 0$, where 0 is the least element of $\mathbb{A}$.

On the other hand, in order to show that a loop or a recursive method does not terminate, we prove that there exist conditions in which its exit points are *unreachable* [5, 7]. Le et al. [6] introduce a specification logic and a verification framework that can specify and verify both terminating and non-terminating behaviors of programs. This logic introduces three temporal predicates Term[R], Loop, and MayLoop to denote definite termination, definite non-termination, and possible non-termination of programs, given that R is a ranking function with a well-ordered co-domain, as defined in Definition 2.2. The semantics of these predicates are defined based on a resource capacity predicate $\mathrm{RC}\langle l, u\rangle$ with the lower and upper bound, in terms of program execution length. That is, a program terminates iff it has a *finite upper bound* on its execution length. On the other hand, the program does not terminate iff the *lower bound* of its execution length is *infinite*. Since this specification logic has a well-defined semantics and is verifiable, specifications in this logic can be embedded into a blockchain to validate the termination and non-termination of programs.

3 OVERVIEW

```
contract Ackermann {
  function ack(int m, int n) returns (int) {
    if (m == 0) return n+1;
    else if (n == 0) return ack(m-1, 1);
    else return ack(m-1, ack(m, n-1));
  }
}
```

Figure 1: The Ackermann smart contract

Consider the above smart contract Ackermann in a Solidity-like language (Figure 1), whose the recursive function ack is an implementation of the well-known Ackermann function[2]. The function does not terminate when the parameter m or n is negative, that is $m < 0 \vee n < 0$. In the other case when $m \geq 0 \wedge n \geq 0$, the function terminates but with a deep level of recursion, even for small inputs. Therefore, without knowing these conditions beforehand, people cannot determine if the program terminates or not by observing its run-time execution. The termination of this function can be proved with the *lexicographic* ranking function $[m, n]$. Such terminating and non-terminating behaviors of the function ack can be specified in a logical specification language and verified [6], as follows.

case {
$m < 0 \vee n < 0 \rightarrow$ requires Loop ensures false;
$m \geq 0 \wedge n \geq 0 \rightarrow$ requires Term$[m, n]$ ensures res $\geq n + 1$; }

The above specification denotes two distinct cases of the program's termination (i.e., $m \geq 0 \wedge n \geq 0$) and non-termination (i.e., $m < 0 \vee n < 0$). In each case, we provide a pair of pre- and post-condition (requires and ensures, respectively), in which the

[2] https://en.wikipedia.org/wiki/Ackermann_function

termination and non-termination of the method are specified in the pre-condition since they encode the initial resource capacity for the method's future execution. The correctness of this specification can be statically verified against the program's source code by the verification framework developed by Le et al. [6]. Moreover, we can also utilize their inference system [7, 8] to analyze the program and derive such specification automatically.

This specification indicates that the method ack may or may not terminate, which depends on the user inputs m and n when this method is invoked. A blockchain system could utilize the information of the derived termination and non-termination conditions in such specifications to execute methods in smart contracts smarter. That is, the system should not allow non-terminating methods to run. On the other hand, terminating methods would be executed with or without resource constraints based on the system's policy. In the next section, we will elaborate our approach to construct a smarter execution model of smart contracts in blockchain systems.

4 A SMARTER EXECUTION MODEL FOR SMART CONTRACTS

In this work, we first derive a specification capturing both terminating and non-terminating behaviors of each function in a smart contract submitted to the blockchain system. These specifications are also recorded into the chain as a metadata of the smart contract. They will be used later to validate user inputs before executing the contract. However, discovering these specifications with termination and non-termination conditions, as well as ranking functions supporting the termination proofs, is not straightforward. Therefore, we utilize available literature works, such as [7], for this specification inference. Based on the meta-information about the termination and non-termination of a smart contract, we propose a new execution model for the smart contract computation in a blockchain system.

Given the current execution context ρ, which captures the user inputs and the internal state of the blockchain. This context can be observed by a real-time monitor integrated inside the blockchain system. Assume that under this context, the system will execute a loop or a method, whose specification is as follows.

$$\mathsf{case}\ \{\ c_i \rightarrow \mathsf{requires}\ P_i\ \mathsf{ensures}\ Q_i\ \}_{i=1}^{n}$$

In this specification, the case conditions $c_1, ..., c_n$ have two properties:

(1) Mutual disjointness: $\forall i \neq j.\, c_i \wedge c_j \Longrightarrow \mathsf{false}$
(2) Completeness: $\mathsf{true} \Longrightarrow c_1 \vee ... \vee c_n$

Theorem 4.1 (Determinism Choice). *Given a set of case conditions $\{c_1, ..., c_n\}$ and a current context ρ. There is* exactly one *case condition c_i which is satisfied by the current context ρ, i.e., $\rho \Longrightarrow c_i$.*

Without the loss of generality, we assume that c_i is the unique case condition which the context ρ satisfies. Consider three scenarios of the pair of pre- and post-condition P_i and Q_i associating with the condition c_i.

Scenario 1 ($P_i \equiv$ Loop). In this scenario, a non-terminating loop or method in the smart contract is being invoked under the current context ρ. Therefore, the blockchain system must prevent this invocation and terminate the execution of the smart contract. This early detection and prevention of non-terminating contract execution also help current resource-constrained systems, like Ethereum, save computation resources uselessly spent on this execution. In this case, the system will throw an exception with the current context and the satisfied non-termination condition as an explanation for this interruption. Moreover, all effects of the current execution have to be reversed, like what happens when an out-of-resource exception occurs in the current resource-constrained blockchain systems.

Scenario 2 ($P_i \equiv \mathsf{Term}[r]$). In this scenario, we should allow this terminating loop or method to run. However, if its execution is suddenly interrupted with an *out-of-resource* exception, the users can be recommended to re-run the contract with more resources by their own risk. Or even better, the blockchain system can rely on the contract profiling of the initial context ρ and the context ρ' at the time when the exception occurs to estimate the needed amount of resources for the contract terminating properly based on the ranking function r.

Scenario 3 ($P_i \equiv$ MayLoop). In a resource-constrained blockchain system, the users can still run this possibly non-terminating program by their own risk and the *out-of-resource* exception may or may not occur. In other systems without any constraints on the execution resources, the invocation of this loop or method is prohibited because it may damage the system's consistency, unless the users can provide an evidence (e.g., a ranking function) that the program terminates.

5 CONCLUSION

In this paper, we propose an initial idea on proving termination and non-termination of smart contracts and how to use them to improve the execution model of the smart contracts in blockchain systems. In future, we would like to implement this proposal and evaluate its efficiency on a real-world blockchain system.

REFERENCES

[1] Vitalik Buterin. 2014. Ethereum: A next-generation smart contract and decentralized application platform. *URL https://github. com/ethereum/wiki/wiki/% 5BEnglish% 5D-White-Paper* (2014).
[2] Christian Cachin. 2016. Architecture of the Hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*.
[3] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. 2008. Proving Conditional Termination. In *CAV*. 328–340.
[4] Nathan Cook. 2017. ZEN: Technical notes on a financial engine. (2017).
[5] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving non-termination. In *POPL*. 147–158.
[6] Ton Chanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin. 2014. A Resource-Based Logic for Termination and Non-termination Proofs. In *ICFEM*. 267–283.
[7] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. 2015. Termination and non-termination specification inference. In *PLDI*. 489–498.
[8] Ton Chanh Le, Quang-Trung Ta, and Wei-Ngan Chin. 2017. HipTNT+: A Termination and Non-termination Analyzer by Second-Order Abduction - (Competition Contribution). In *TACAS*. 370–374.
[9] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf. (2008).
[10] Pierre-Yves Strub, Nikhil Swamy, Cédric Fournet, and Juan Chen. 2012. Self-certification: bootstrapping certified typecheckers in F* with Coq. In *POPL*. 571–584.
[11] Nick Szabo. 1997. The idea of smart contracts. *Nick Szabo's Papers and Concise Tutorials* (1997).
[12] Alan Turing. 1989. The Early British Computer Conferences. MIT Press, Cambridge, MA, USA, Chapter Checking a Large Routine, 70–72.