Linear logic as a partly-executable specification language for smart contracts

Denis Erfurt
Jack Ek

ABSTRACT

asd asdas

1 INTRODUCTION

We propose to use a *logical programming language* based on linear logic to specify smart contracts. For those unfamiliar with the logical programming paradigm, it means that the programmer uses logical propositions to specify *what* the program should do or *which constraints* it should satisfy, but not *how* this is achieved. From this, the computer creates a suitable program and executes it. In short, in logical programming, the specification is executable and the program is correct-by-construction, although there are caveats to this interpretation which we will discuss later.

For those coming from a typed functional programming background, it can help to think of a logical language as a typed functional language, but in which the programmer only writes type signatures, no functions. This understanding is based directly on the Curry-Howard correspondence, which essentially states that type signatures are the same things as logical propositions, and programs satisfying type signatures are the same things as proofs of those propositions.

Traditionally, logical programming is based on the notion of *proof search*, in which the execution of a program is defined as finding a proof of a particular proposition. (Again, this can be understood as *program synthesis*, where the computer automatically creates a program which satisfies a particular constraint.) Unfortunately, proof search is undecidable for the full linear logic, so proofs will have to be written manually in some cases. And even in the cases where it is decidable, it can be a fairly expensive procedure which is arguably not viable for smart contracts due to the cost of running execution under consensus. As we will see however, there exist alternative execution strategies, which are remarkably well-suited for executing smart contracts.

2 MOTIVATION

There are several reasons for why we think a logical language based on linear logic is a promising approach when it comes to smart contract programming.

- Formal logics is an old and well-studied family of mathematical systems.
- Linear logic can be interpreted as a logic concerning itself with *state* and *scarce resources*, both of which are central to smart contracts.
- Proofs in linear logic can be given several different Curry-Howard correspondences, granting programmers the flexibility to write proofs/programs in several different programming languages, depending on which one is best suited. For example:
  - Terms in the π-calculus can be used as linear logical proofs. The π-calculus is one of the most well-studied concurrent computational models. If blockchain sharding becomes reality, concurrency will almost certainly play a central role, thus making languages that can deal with it in a native and principled way ideal candidates for blockchain languages.
  - In the same way, terms in the λ-calculus can be used, granting programmers a rather strict programming interface for when they actively don't want to consider the complexities of concurrent programs.

2.1 Requirements

R01 Programming obvious applications has to be intuitive.
R02 Practically proving the correctness of programs must be possible.
R03 Execution of compiled programs must be relatively efficient.
R04 Programs have to be composable with other programs (i.e. programs are used to build larger programs, are used to build larger programs).

3 GENERAL IDEA

The general Idea is to view the blockchain [contract] as a multiset $\mathcal{B}$ of resources.

$$\frac{\dfrac{\mathcal{D}}{\Delta \vdash C} \quad \exists\theta.\Delta\theta \subseteq \mathcal{B}}{\mathcal{B} \longrightarrow \mathcal{B} \setminus \Delta\theta \cup \{C\theta\}}\,\mathcal{B}\text{-transition} \tag{1}$$

3.1 Unification

During a blockchain transition step a Most General Unifier(MGU) θ has to be found. Initially θ contains the assigment of reserved variables, such as $Sender$, $Origin$, $BlockNumber$, etc, which don't change during unification.

4 EXAMPLE PROGRAMMS

Lets consider a simple token.

5 SECURITY

We are interested to proof different properties about our written programs. E.g. that the total value of a system is not changed with different operations, this we want to illustrate on a case study - a coin exchange: Let d denote a dime, n a nickel and q a quarter together with the following persistent rules given:

$$\left.\begin{array}{l} r1 : q \multimap d \otimes d \otimes n \\ r2 : d \otimes d \otimes n \multimap q \\ r3 : d \multimap n \otimes n \\ r4 : n \otimes n \multimap d \end{array}\right\} R$$

Also we want to have a persistent notion of value:

$$\left.\begin{array}{l} v1 : q \multimap value(25) \\ v2 : d \multimap value(10) \\ v3 : n \multimap value(5) \end{array}\right\} V$$

Additionally we can add up values:

$$sum : value(X) \otimes value(Y) \multimap value(X + Y)$$

Now our task is to prove the stability of the system. This means that the total value don't change if a rule in R is applied. We can do this easily by proof expansion of every rule in R. We want to illustrate this for $r1$, the other cases can be shown analogous:
Without loss of generality let $\Delta = value(X), q$, we want to proof $\Delta \vdash value(X + 25)$ first by applying only Rules out of V + sum as seen in Proof 1 and expand this proof by applying $r1$ first and then restricting ourselves with only using rules out of V, which can be seen in Proof 2. This proves that the application of rule $r1$ didn't change the total value. With this we have shown that rules in R satisfy our stability criteria. The proves are rather technical and included for the purpose to demonstrate, that those proves can be easily automatized.

TODO: provided that the computation is deterministic and arbitrary values can't be derived. Find a way how to prove those properties automatically.

6 EFFICIENCY

As computation is done via proof-search and unification, executing programms tend to be slow in practice. Therefore we are interested in methods which reduces the computational overhead. One such method could be the recuction of proof search to the execution of a minimal set of evm instructions, which yield the same result. For this the equivalence of the proof search to the evm instructions has to be **proven**.

In order to achieve this, we will reconstruct the evm in linear logic and show for a proposition A, a proposition B which represents an evm program and a substitution θ:

$$\Gamma; \cdot \vdash A(\bar{X})\theta \Leftrightarrow \Gamma; \cdot \vdash B(\bar{X})\theta$$

TODO: Or is it $\Gamma; \cdot \vdash A(\bar{X}) \multimap\!\!\circ\, B(\bar{X})$?

Meaning that the evm programm B is behaviourally isomorphic to A.

EVM implementation

For demonstration purposes we will just implement a basic stack machine here, which should be sufficient to illustrate the approach.

The EVM has a few distinct components, such as:

- code
- pc
- stack
- ...

We can model the Stack Machine with the following predicates:

- code(X,Y) - OpCode Y is on position X in the code.
- pc(X) - Program Counter is at position X
- stack(X,Y) - word Y is on position X on the stack
- ...

Additionally we create a few extra predicates which will help during the execution:

- stackLength(X) - X is the current stack length

We now want to model the predicate $inc(X, Y) := X + 1 = Y$ in the evm with the following program: $PUSH\ X\ INC$
The result should then be found at stack height 0. We can retrieve it with:

$$stack(0, X) \otimes pc(3) \multimap result(X)$$

Also initially we have the following predicates given: $pc(0) \otimes stackLength(0)$

PUSH

$$pc(X) \otimes code(X, PUSH) \otimes code(sX, V) \otimes stackLength(L)$$

$$\multimap$$

$$stack(L, V) \otimes pc(ssX) \otimes code(X, PUSH) \otimes code(sX, V) \otimes stackLength(sL)$$

This pushes the value V to the stack, increments the stack length and preserves the code.

INC

Assume we have the following axiomatic rule for inc given:

- inc(X, sX)

Thus:

$$inc(X, Y) \Leftrightarrow X + 1 = Y$$

So the following transition rule models our INC opCode:

$$pc(X) \otimes code(X, INC) \otimes stack(L, V) \otimes stackLength(sL) \otimes inc(V, W) \multimap pc(sX) \otimes code(X, INC) \otimes stackLength(sL) \otimes stack(L, W)$$

proof

Together we have to prove:

$$inc(X, Y) \multimap\!\!\circ\ (C \otimes \mathcal{R} \otimes pc(0) \otimes stackLength(0) \multimap result(Y))$$

with the code:

$$C := code(0, PUSH) \otimes code(1, X) \otimes code(2, INC)$$

and the termination condition:

$$\mathcal{R} := pc(3) \otimes stackLength(1) \otimes stack(0, Z) \multimap result(Z)$$

TODO: Make the actual proof, but the conjecture is that this is proovable.

TODO: How does this work with non-primitives, e.g. loops, math-function e.g. log, exp

7 APPENDIX

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{value(X) \otimes value(25) \vdash value(Y) \otimes value(Y')}\,id(\sigma) \quad \dfrac{}{value(Y+Y') \vdash value(X+25)}\,id(\sigma)}{value(X) \otimes value(25), value(Y) \otimes value(Y') \multimap value(Y+Y') \vdash value(X+25)}\,\multimap L}{value(X) \otimes value(25) \vdash value(X+25)}\,copy_{sum}}{value(X), value(25) \vdash value(X+25)}\,\otimes L \quad \dfrac{}{q \vdash q}\,\mathrm{id}}{\dfrac{value(X), q, q \multimap value(25) \vdash value(X+25)}{value(X), q \vdash value(X+25)}\,copy_{v1}}\,\multimap L$$

$$\sigma = \{\tfrac{X}{Y}, \tfrac{25}{Y'}\}$$

Abbildung 1: Proof using only rules from V

$$\dfrac{\dfrac{}{q \vdash q}\,\mathrm{id} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{d \vdash d}\,\mathrm{id} \quad \dfrac{\dfrac{\mathcal{D}}{value(Y+Y') \otimes value(10), n \vdash value(X+25)}\,copy_{sum}}{value(Y+Y'), value(10), n \vdash value(X+25)}\,\otimes L}{\dfrac{value(Y+Y'), d, d \multimap value(10), n \vdash value(X+25)}{value(Y+Y'), d, n \vdash value(X+25)}\,copy_{v2}}\,\multimap L \quad \dfrac{}{value(X) \otimes value(10) \vdash value(Y) \otimes value(Y')}\,id(\sigma)}{value(X) \otimes value(10), value(Y) \otimes value(Y') \multimap value(Y+Y'), d, n \vdash value(X+25)}\,\multimap L}{value(X) \otimes value(10), d, n \vdash value(X+25)}\,copy_{sum}}{value(X), d, n, value(10) \vdash value(X+25)}\,\otimes L \quad \dfrac{}{d \vdash d}\,\mathrm{id}}{value(X), d, d, n, d \multimap value(10) \vdash value(X+25)}\,\multimap L}{value(X), d, d, n \vdash value(X+25)}\,copy_{v2}}{value(X), d, d \otimes n \vdash value(X+25)}\,\otimes L}{value(X), d \otimes d \otimes n \vdash value(X+25)}\,\otimes L}{\dfrac{value(X), q, q \multimap d \otimes d \otimes n \vdash value(X+25)}{value(X), q \vdash value(X+25)}\,copy_{r1}}\,\multimap L$$

With $\mathcal{D}$:=

$$\dfrac{\dfrac{}{value(Y+Y') \otimes value(10) \vdash value(Z) \otimes value(Z')}\,id(\sigma) \quad \dfrac{\dfrac{\dfrac{\mathcal{E}}{value(Z+Z'), value(5) \vdash value(X+25)}\,\otimes L \quad \dfrac{}{n \vdash n}\,\mathrm{id}}{value(Z+Z'), n, n \multimap value(5) \vdash value(X+25)}\,\multimap L}{value(Z+Z'), n \vdash value(X+25)}\,copy_{v3}}{value(Y+Y') \otimes value(10), value(Z) \otimes value(Z') \multimap value(Z+Z'), n \vdash value(X+25)}\,\multimap L$$

With $\mathcal{E}$:=

$$\dfrac{\dfrac{\dfrac{}{value(Z+Z') \otimes value(5) \vdash value(Q) \otimes value(Q')}\,id(\sigma) \quad \dfrac{}{value(Q+Q') \vdash value(X+25)}\,id(\sigma)}{value(Z+Z') \otimes value(5), value(Q) \otimes value(Q') \multimap value(Q+Q') \vdash value(X+25)}\,\multimap L}{value(Z+Z') \otimes value(5) \vdash value(X+25)}\,copy_{sum}$$

With $\sigma := \{\tfrac{X}{Y}, \tfrac{10}{Y'}, \tfrac{X+10}{Z}, \tfrac{10}{Z'}, \tfrac{X+10+10}{Q}, \tfrac{5}{Q'}\}$

Abbildung 2: Proof using rule $r1$ and only rules from V