Findel: Secure Derivative Contracts for Ethereum

Alex Biryukov[1], Dmitry Khovratovich[2], Sergei Tikhomirov[2]

[1] alex.biryukov@uni.lu
University of Luxembourg
[2] {khovratovich,sergey.s.tikhomirov}@gmail.com
University of Luxembourg

Abstract. Blockchain-based smart contracts are considered a promising technology for handling financial agreements securely. In order to realize this vision, we need a formal language to unambiguously describe contract clauses. We introduce Findel – a purely declarative financial domain-specific language (DSL) well suited for implementation in blockchain networks. We implement an Ethereum smart contract that acts as a marketplace for Findel contracts and measure the cost of its operation. We analyze challenges in modeling financial agreements in decentralized networks and outline directions for future work[3].

Keywords: blockchain, smart contracts, financial engineering, domain-specific language

1 Introduction

Financial derivatives – contracts defined in terms of other contracts – play a major role in modern economy[4]. Financial industry lacks a universal domain-specific language. Natural language is unsuitable for expressing contracts due to its inherent ambiguity. An influential paper [JES00] is one of many attempts to create a rigorous DSL that would mitigate disputes and stimulate automated processing of complex derivatives. It leverages ideas from functional programming and uses a succinct set of basic building blocks to express financial agreements. A key feature of this notation is composability: new indefinitely complex derivatives can be defined based on existing ones. Due to their nested structure, contracts in this DSL are well-suited for automated processing, including valuation. The authors do not specify an enforcement mechanism though: execution is performed by an implicit environment. This work forms the basis for research [Gai11] [Sch14] and commercial [FSNB09] [Mor16] projects.

[3] See the author's post-print at https://orbilu.uni.lu/handle/10993/30975 and the related source code at https://github.com/cryptolu/findel.

[4] The derivatives market is comparable in size to the world's GDP. The gross market value of all outstanding over-the-counter derivatives is $20.7 trillion [Bis16] (2016). The world GDP in 2015 is $73,9 trillion [Wor16].

The idea of smart contracts – computer programs for (semi-)automatic enforcement of agreements – dates back to mid-1990s [Sza97]. Blockchain networks, notably Ethereum, became the first practical implementation of this idea and fueled interest in the concept [dC16]. Ethereum is a network of mutually distrusting nodes, which nevertheless establish consensus on the results of computations without the need of a trusted third party.

An obvious use case for blockchain-based smart contracts is to securely manage financial agreements. A naive approach to doing so is to encode the entire logic of an agreement inside a smart contract. Expressing complex clauses in a general-purpose programming language, like Ethereum's Solidity, is error-prone [ABC16] [Sir16]. We propose a safer approach that separates the description of a contract from its execution. A user only defines what a contract *is* ("I owe you $10 tomorrow"), not *how* it is executed ("if the timestamp is greater than t_0, ..."). The entire execution logic is implemented inside a smart contract, which is executed by nodes of a blockchain network. Thus we take the best of both worlds: unambiguity and composability of a concise declarative DSL, and trustless execution of blockchain-based smart contracts.

We introduce **Findel** (Financial Derivatives Language) – a declarative financial DSL (Section 2) capable of expressing most common derivatives (Appendix A). We implement an Ethereum contract that manages Findel contracts (Section 3) and prove our approach viable in terms of cost (Section 4).

2 Findel contracts syntax

2.1 Definitions

Definition 1. *A **Findel contract**[5] C is a tuple (D, I, O), where D is the **description**, I is the **issuer**, and O is the **owner** (collectively called parties).*

Definition 2. *A **description** of a Findel contract is a tree with **basic primitives** as leaves and **composite primitives** as internal nodes. The following BNF grammar defines primitives:*

⟨*basic*⟩ ::= `Zero` | `One (` ⟨*currency*⟩ `)`

⟨*scale*⟩ ::= `Scale (` ⟨*number*⟩ `,` ⟨*primitive*⟩ `)`

⟨*scaleObs*⟩ ::= `ScaleObs (` ⟨*address*⟩ `,` ⟨*primitive*⟩ `)`

⟨*give*⟩ ::= `Give (` ⟨*primitive*⟩ `)`

⟨*and*⟩ ::= `And (` ⟨*primitive*⟩ `,` ⟨*primitive*⟩ `)`

⟨*or*⟩ ::= `Or (` ⟨*primitive*⟩ `,` ⟨*primitive*⟩ `)`

⟨*if*⟩ ::= `If (` ⟨*address*⟩ `,` ⟨*primitive*⟩ `,` ⟨*primitive*⟩ `)`

[5] We may refer to Findel contracts simply as contracts, when the distinction between them and Ethereum smart contracts is clear from the context.

⟨*timebound*⟩ ::= `Timebound` (⟨*timestamp*⟩ , ⟨*timestamp*⟩ , ⟨*primitive*⟩)

⟨*composite*⟩ ::= ⟨*scale*⟩ | ⟨*scaleObs*⟩ | ⟨*give*⟩ | ⟨*and*⟩ | ⟨*or*⟩ | ⟨*if*⟩ | ⟨*timebound*⟩

⟨*primitive*⟩ ::= ⟨*basic*⟩ | ⟨*composite*⟩

We distinguish between composite and basic primitives, because the former contain other primitives as sub-nodes while the latter do not. *Currency*, *number*, *address*, and *timestamp* are implementation dependent data types. D and I can not be modified after a contract is created.

A financial company typically has templates for common contracts. Parties who wish to sign an agreement write their names on a copy of a template and sign it, making it unique and legally binding. In our model, Findel contracts represent signed copies while their descriptions represent blank templates.

Traditional contracts usually contain clauses that regulate sub-ideal situations, i.e., a breach of contract. Findel does not distinguish between "ideal" and "sub-ideal" situations. All right and obligations are expressed uniformly. Section 3.3 discusses issues related to contract enforcement.

Table 1 informally defines the primitives' execution semantics.

Primitive	**Informal semantics**
Basic	
Zero	Do nothing.
One(*currency*)	Transfer 1 unit of *currency* from the issuer to the owner.
Composite	
Scale(k, c)	Multiply all payments of c by a constant factor k.
ScaleObs($addr, c$)	Multiply all payments of c by a factor obtained from $addr$.
Give(c)	Swap parties of c.
And(c_1, c_2)	Execute c_1 and then execute c_2.
Or(c_1, c_2)	Give the owner the right to execute either c_1 or c_2 (not both).
If($addr, c_1, c_2$)	If b is true, execute c_1, else execute c_2, where b is a boolean value obtained from $addr$.
Timebound(t_0, t_1, c)	Execute c, if the current timestamp is within $[t_0, t_1]$.

Table 1. Findel contract primitives

Table 2 illustrates the composability of Findel[6].

2.2 Execution model

Findel contracts have the following lifecycle:

[6] INF is a symbol representing infinite time, i.e., $t_0 < INF$ for every t_0. δ is an implementation dependent constant intended for handling imperfect precision of time signal in distributed networks.

Contract	Definition
$\text{At}(t_0, c)$	$\text{Timebound}(t_0 - \delta, t_0 + \delta, c)$
$\text{Before}(t_0, c)$	$\text{Timebound}(now, t_0, c)$
$\text{After}(t_0, c)$	$\text{Timebound}(t_0, INF, c)$
$\text{Sell}(n, CURR, c)$	$\text{And}(\text{Give}(\text{Scale}(n, \text{One}(CURR))), c)$

Table 2. Examples of custom Findel contracts

1. The first party **issues** the contract by specifying D, becoming its issuer. This is a mere declaration of the issuer's desire to conclude an agreement and entails no obligations.
2. The second party **joins** the contract, becoming its owner. As a result, both parties accept certain rights and obligations.
3. The contract is **executed** immediately as follows:
   (a) Let the root node of the contract's description be the current node.
   (b) If the current node is either `Or` or `Timebound` with $t_0 > now$, postpone the execution: issue a new Findel contract with the same parties and the current node as root. The owner can later demand its execution.
   (c) Otherwise, execute all sub-nodes recursively[7].
   (d) Delete the contract.

The execution outcome is fully determined by description D, execution time t, and external data $\mathcal{S}$ retrieved at time t.

2.3 Example

Suppose Alice sells to Bob a zero-coupon (i.e., paying no interest) bond that pays \$11 in one year for \$10:

$$c_{zcb} = \text{And}(\text{Give}(\text{Scale}(10, \text{One}(USD))), \text{At}(\text{now+1 years}, \text{Scale}(11, \text{One}(USD))))$$

We now show how c_{zcb} is executed step by step.

1. And executes; Bob temporarily owns two new contracts:

Alice's contracts	
Alice's balance	100
Bob's contracts	$\text{Give}(\text{Scale}(10, \text{One}(USD)))$ $\text{At}(\text{now} + 1 \text{ years}, \text{Scale}(11, \text{One}(USD)))$
Bob's balance	10

2. Give executes; Alice owns a new contract:

Alice's contracts	$\text{Scale}(10, \text{One}(USD))$
Alice's balance	100
Bob's contracts	$\text{At}(\text{now} + 1 \text{ years}, \text{Scale}(11, \text{One}(USD)))$
Bob's balance	10

[7] In case of `Or`, execute exactly one of the sub-nodes, according to the owner-submitted value indicating the choice; delete the other one. It is the only primitive that requires an additional user-supplied argument for execution.

3. Scaled One transfers $10 go from Bob to Alice:

Alice's contracts	
Alice's balance	110
Bob's contracts	$\mathrm{At}(\mathrm{now} + 1 \text{ years}, \mathrm{Scale}(11, \mathrm{One}(USD)))$
Bob's balance	0

4. In one year Bob claims $11 from Alice:

Alice's contracts	
Alice's balance	99
Bob's contracts	
Bob's balance	11

3 Implementation

We develop an Ethereum smart contract, referred to as marketplace, that keeps track of users' balances and lets them create, trade, and execute Findel contracts. The Findel DSL is network-agnostic and can be implemented on top of any blockchain with sufficient programming capabilities.

3.1 Ethereum overview

Ethereum is a decentralized smart contracts platform [But14] [Woo14]. Ethereum full nodes store data, perform computations, and maintain consensus about the state of all accounts using a proof-of-work mechanism similar to that in Bitcoin. Programs (Ethereum smart contracts) are stored on the blockchain as Ethereum virtual machine (EVM) bytecode, a Turing-complete language. Programmers write contracts in high-level languages targeting EVM, most popular being Solidity and Serpent (we use the former).

A contract can call other contracts' functions and send them units of Ether – the Ethereum native cryptocurrency. To launch a particular function of a contract, a user must send a well-formed transaction to the Ethereum network.

Each EVM operation has a fixed cost in *gas*. A user pays upfront for the maximum amount of gas the computation is expected to consume and gets a partial refund after a successful execution. If an exception (including "out of gas") occurs, all changes are reverted, but the gas is not refunded.

3.2 Implementation details

Users and balances We implement the objects defined in Section 2.1 with struct data types `Description` and `Fincontract`. We also introduce the `User` type that contains the user's Ethereum address and balances in all supported currencies. Users, descriptions and contracts are stored in their respective mappings (a generic key-value storage type in Solidity) in the marketplace's storage.

The ultimate effect of every financial agreement is changing the parties' balances (with clauses specifying when and under what conditions it should occur). We stick to a naive approach: each user is assigned an array of balances for each

supported currency. Although easily implementable, it introduces a single point of failure: the marketplace holds users' deposits.

The only primitive that actually transfers value is One. The `enforcePayment` function implements its execution. It subtracts a given amount in a given currency from the issuer's balance and adds it to the owner's balance. Our current implementation does not enforce any constraints on users' balances that would prevent them from building up too much debt.

Ownership transfer In addition to `issuer` and `owner` (see Definition 1), a `Fincontract` contains an auxiliary `proposedOwner` field. On contract creation, `issuer`, `owner`, and `proposedOwner` are initialized to `msg.sender`. To transfer ownership, the owner sets `proposedOwner` either to the address of the proposed new owner or to `0x0`. Only the proposed owner can (but does not have to) `join` the contract; `0x0` means anyone can do so[8].

Data sources and gateways Ethereum contracts are intentionally isolated from the broader Internet and can not pull data from the Web, as it can not be consistently replicated [Gre16]. Asynchronous requests usually solve the problem: a smart contract records an Ethereum event with request parameters properly encoded. A daemon process at an Ethereum node listens for such events, parses requests, and sends them to the Web. The responses are then sent to the requesting smart contract on behalf of an Ethereum account affiliated with the daemon. The submitted data may be accompanied by a proof of authenticity (say, digital signature on a pre-approved public key)[9].

Financial derivatives often use external data. To prevent a malicious or careless user from creating a Findel contract using untrusted sources, we need to guarantee data authenticity.

Definition 3. *A **gateway** is a smart contract that conforms to the API:*

- ***int getValue()*** *Get the latest observed value*[10]*.*
- ***uint getTimestamp()*** *Get the timestamp at which the latest value was observed.*
- ***bytes getProof()*** *Get the authenticity proof for the latest value.*
- ***update()*** *Update the value.*

[8] Beware of front-runners: Bob can monitor the network and try to join a contract as soon as he sees Alice's attempt to do so. Depending on the network latency and miner's behavior, either transaction can be confirmed.

[9] BTCRelay is a prominent example: users submit Bitcoin block headers to a smart contract, which implies their authenticity from the validity of easily verifiable proof-of-work. After a header is stored on the Ethereum blockchain, users check with a Merkle proof that the Bitcoin block contains a given transaction.

[10] For simplicity, we only consider 256-bit integers as observable values. Boolean values can be trivially simulated via integers.

A gateway connects to an external data source and stores the latest value observed along with the time of observation, and, optionally, a cryptographic proof of authenticity. We do not specify the type of proof a gateway provides. Possible options include Oraclize [Ora16] / TLSNotary [Tls16] and Reality Keys [Rea16].

The marketplace queries a gateway at execution time, if necessary. If the value is fresh and the proof is valid, the execution proceeds, otherwise it is aborted and all changes are reverted. Since a Findel contract may use multiple gateways, the owner is advised to `update` them all shortly before execution.

A possible improvement would be for a gateway to store not only the latest observed value, but a sequence of historical data. This would allow for more straightforward modeling of derivatives that depend on multiple data points, such as barrier options (execute either c_1 or c_2 depending on whether an observable value touches a pre-defined threshold between acquisition and maturity).

We assume that the original data sources (e.g., feeds of reputable financial media) are trustworthy. An extra safety catch would be to query multiple sources, exclude outliers and return an aggregated value. Authenticity of data sources is guaranteed by a secure connection (e.g., TLS) and the existing PKI for authentication ([CF14] and [LC16] propose blockchain-based PKI architectures).

Gateways without publicly available source code should not be trusted.

Execution implementation The `executeRecursively` function implements the execution logic defined in Section 2.2 and returns `true` if executed completely (without creating new contracts) and `false` otherwise. The execution of an expired contract ($t_0 < now$) returns `true` unconditionally[11] and deletes the contract[12]. Every step in the life cycle of a Findel contract issues a system-wide notification (`Event`), allowing users to keep track of contracts they are interested in.

Our implementation deviates from the model (Section 2.1) in that the execution of contracts is not guaranteed. Ethereum contracts can not act on their own: the owner must issue a transaction to trigger execution. The owner may be unable to do so due to either opportunistic behavior, or technical problems, such as loss of connectivity or lack of ether. Thus we presume that Findel contracts are not guaranteed to execute[13]. We discuss this issue in Section 3.3.

We model unbounded Findel contracts (i.e., with INF as the upper time bound) using a global *expiration* constant inside the marketplace contract. Every Findel contract in the Ethereum implementation can only be executed within *expiration* time units after creation (e.g., 10 years).

[11] By definition, an expired contract is equivalent to Zero.

[12] An expired contract should also be deleted even if its owner is offline forever. Our current implementation does not handle the latter case, though it may be considered an attack vector due to increasing storage usage. A possible approach is for a marketplace to offer rewards for keeping track of expired contracts and triggering their deletion.

[13] Compare to [JES00]: "If you acquire *(c1 or c2)* you must immediately acquire either *c1* or *c2* (but not both)". We can not force a user to make this decision.

3.3 Possible improvements

We now discuss the shortcomings of our model and ways to improve it.

Enforcement As mentioned in Section 3.2, Findel contracts are not guaranteed to execute. At first sight, it is a major problem, as contract must impose obligations on parties. In traditional finance, a trusted third party and, ultimately, the state law enforcement are responsible for punishing violators. The closest we can arguably get to enforcement is a conditional penalty implemented inside a Findel contract itself.

Assume Alice issues and Bob joins the following contract:

$$C = Before(t_0, \text{Or}(\text{Give}(\text{One}(USD)), \text{Give}(\text{One}(EUR))))$$

C obliges Bob to give Alice either \$1 or €1 before time t_0. If Bob fails to make a choice on time, Alice does not get the money she was planning to receive[14]. To prevent it, Alice attaches a "penalty" clause:

$$P = After(t_0, \text{If}(c_{executed}, \text{Zero}, \text{Scale}(2, \text{One}(USD))))$$

$c_{executed}$ is the address of a gateway that indicates whether a particular Findel contract was executed. When Bob joins $C_{penalty} = \text{And}(C, \text{Give}(P))$, Alice obtains the right to claim \$2 from Bob if he fails to fulfil his obligations.

Note that $C_{penalty}$ references $C_{executed}$, which in turn must be aware of $C_{penalty}$. Thus the gateway should be either adjustable (with Alice tuning the gateway with a special transaction) or generic (reports the state of a Findel contract taking its id as an argument).

Defaulting on debt A concise financial DSL does not prevent borrowers from defaulting on their debt. It is up to a marketplace to solve this problem.

Requiring a 100% guarantee deposit seems safe, but is questionable from an economical standpoint. People and organizations borrow money to invest it. The no-arbitrage principle states that there is no guaranteed way to make a profit. The investor reward, e.g. interest, is the premium for taking the inevitable risk of business failure. Thus, this approach hardly makes economical sense.

A marketplace can also mimic the fractional reserve banking model by requiring users to always be able to pay at least n% of their debt and punishing violators (e.g., by withholding their guarantee deposit). It does not solve the problem of defaults completely though. In legacy finance, users have a fixed government-issued identity, allowing banks to maintain a common database of their credit history. In a decentralized setting, users can create a practically indefinite number of identities. A production-ready marketplace should therefore take measures to combat Sybil attacks.

[14] In this particular case, an equivalent contract $\text{Give}(\text{Or}(\text{One}(USD), \text{One}(EUR)))$ solves the issue. In more complex cases this is not necessarily the case.

Modeling balances with Tokens A more refined approach to modeling users' balances is to use **tokens** – a de-facto standard API [Tok16] for implementing transferable units of value in Ethereum. Tokens are primarily used to represent company shares during so-called initial coin offerings [Ico17]. We assume that tokens can be freely exchanged to any currency the marketplace operates with. Given the address $\mathcal{T}$ of the Ethereum token contract, any Ethereum contract can query the balance of any user U, and transfer its tokens (if it has any) to an arbitrary address. Suppose Alice and Bob are token holders. Alice calls a standard API function `approve` to allow Bob to withdraw a certain amount of tokens from her account. Bob later calls `transferFrom` to transfer the tokens. The transfer succeeds if Alice has enough funds.

We suggest the following procedure. A Findel contract's issuer approves the marketplace with the number of tokens he is potentially liable with. The marketplace implements `enforcePayment` by calling `transferFrom` thus trying to withdraw tokens from the issuer and send them to the owner. Certainly, for the execution to complete, the owner must either have enough tokens in the account, or execute another Findel contract to fill it up. Thus we delegate the banking functionality to the token smart contract and free the marketplace from holding and transferring money [Kho16].

Multi-party contracts We might want to extend the Findel contracts model to support more than two parties. An example of a three-party contract is buying a car with insurance. A user can only buy a car while simultaneously signing an insurance contract. We can express the two contracts (buyer – car dealer, buyer – insurance company) in Findel DSL, but executing them atomically is non-trivial. A possible way would be to use a gateway that keeps track of the state of Findel contracts. If *insuranceSigned* indicates whether a user joined the insurance contract, then buying with insurance looks like this (assuming CAR is a token representing the ownership over a car):

$$\mathrm{If}(\mathit{insuranceSigned}, \mathrm{And}(\mathrm{Give}(\mathrm{Scale}(P, \mathrm{One}(USD)), \mathrm{One}(CAR))), \mathrm{Zero})$$

Local client In order to communicate with a Findel marketplace, users need client-side software. Besides communicating with the Ethereum network, it might also implement other functions:

- Create and store Findel contracts locally.
- Calculate the current value and other properties of Findel contracts based on assumptions about external data (e.g., the € / $ exchange rate is between 1.0 and 1.2) or valuation techniques such as the lattice binomial model [CRR79].
- Keep track of relevant Findel contracts and perform actions depending on their state (e.g., if c_1 gets executed, join c_2).
- Store a predefined list of addresses of trusted gateways, similar to a list of trusted certificate authorities in web browsers.

3.4 Platform limitations

A Turing-complete programming language does not mean that all a programmer can think of can be implemented inside an Ethereum contract. Gas costs aside, the Ethereum network architecture implies certain limitations.

Lack of precise clock Timing is important for almost all financial contracts. Clock synchronization is a hard problem in decentralized systems, even more so if participants can profit from manipulating timestamps. Blocks in Ethereum are produced every 15 seconds; block timestamps provide causal ordering. Solidity contains keywords for time units, but timestamps are ultimately controlled by miners.

Imperative paradigm Functional programming paradigm is well suited for developing embedded DSLs [Gib13]. The original papers by Peyton Jones et al. as well as all existing implementations of their DSL use functional languages (Haskell [JES00] [JE03] [vS07], OCaml [Lex00], Scala [Wal12] [Cha15]). In contrast, Solidity and Serpent are imperative. Functional languages for Ethereum are in a very early stage of development [FpE17].

Underdeveloped type system Ethereum supports neither decimal nor floating-point types[15], which often model amounts of money and currency exchange rates respectively. The only numeric data types in Solidity are integers of various bit lengths. Moreover, Solidity lacks type parameters, which could be useful for Gateways (i.e., `Gateway<int>`).

4 Gas costs

Every computational step in Ethereum is charged in terms of gas. Despite the use of expensive permanent storage operations, the cost of running our implementation is not prohibitively high for a proof-of-concept.

We measure gas costs of managing common Findel contracts as assessed by the Browser-solidity compiler [Bro16][16] for a marketplace supporting two currencies (referred to as USD and EUR and not tied to any asset). The difference between transaction and execution cost is that the former includes the overhead of creating a transaction (i.e., a call from a client) and the latter does not (i.e., a call from another contract) [Rev16].

4.1 Setup and helper functions

Registering a user implies initializing the user's balances to zero for all supported currencies. For testing purposes, we implement a gateway that uses the current timestamp as data source and calculates a single `keccak256` hash as a dummy authenticity proof.

[15] A likely rationale: rounding issues break consensus.
[16] Solidity version: 0.4.4+commit.4633f3de.Emscripten.clang

Operation	Transaction cost	Execution cost
Create a marketplace smart contract	2221599	1681095
Register a user	79462	58190
Check user's balance	47667	26395
Get contract info	24407	959
Get description info	24706	1258
Update a gateway	36922	15650

Table 3. Cost of setup and helper functions (in gas units)

4.2 Managing common derivatives

In our measurements, we omit cases where parties split the execution cost. We assume that the issuer only pays for contract creation and issuance whereas the owner pays for the execution. For simple Findel contracts, two Ethereum transactions (one from each party) represent the whole lifecycle of a Findel contract. In more complex cases, when a contract executes in multiple steps, we sum up all costs that the owner bears to execute it completely. We also do not account for gateway update costs.

Operation	Create and issue		Join and execute	
	Tx cost	Exec cost	Tx cost	Exec cost
One	184239	177967	58493	93602
Currency exchange (fixed rate)	663149	656877	101878	138430
Currency exchange (market rate)	300842	294570	59822	96196
Zero-coupon bond	373783	367511	143891	201750
Bond with two coupons	939566	933294	346871	477100
European option	519628	513356	278191	411103
Binary option	402359	396087	59826	96204

Table 4. Cost of handling Findel contracts for common derivatives (in gas units)

As of January 2017, the gas cost 10^{-9} ether per unit [Eth17]; the price of ether fluctuated around $10 [Wor17]. That brings the cost of a typical Findel contract operation (10^5 – 10^6 gas units) to 1.8 – 18 US cent.

5 Related work

[Sch13] and [Hvi10] review financial DSLs and related projects. [STM16] and [CBB16] explore approaches to smart contract programming languages.

5.1 Composable contracts by Peyton Jones et al.

Our work is inspired by the composable contracts as defined in [JE03], from which we borrow some of our primitives (Zero, One, Scale, And, Or). It turns out

though that this notation is not directly transferable to blockchain environments (at least to Ethereum) due to the way it formalizes temporal conditions (*when*, *until*). Blockchains differ substantially from traditional centralized marketplaces in how they model conditions. For this reason we introduced If and Timebound primitives to express causal and temporal conditions respectively.

5.2 Logic Portfolio Theory by Steffen Schuldenzucker

Steffen Schuldenzucker in [Sch16] proposes an axiomatic approach to proving no-arbitrage relationships between contracts based on the notation from [JE03]. Using a rigorously defined algebra of contracts, he proves well-known financial theorems, such as the put-call parity. Formal semantics of Findel can be introduced using a similar approach. This would enable formal verification techniques that could substantially increase confidence in the safety of our language.

5.3 Preliminary draft by Nick Szabo

Smart contracts pioneer Nick Szabo in [Sza02] presents "a mini-language" that can be characterized as a middle ground between programming and legal speak. The basic building block is a *right* (e.g., to receive $100 now). Rights are combined using well-defined operators (*when*, *then*, *also*, *with* – analogous to our primitives) and *performed* depending on external events. Parties are assumed to have a trusted source of real-world information. The language is not purely declarative: contracts may perform calculations and save values in state variables, which allows for more flexibility[17].

6 Conclusion

Smart contracts in public blockchain networks seem to be a perfect match for modeling financial agreements. Their unique value proposition is trustless execution, which reduces counterparty risks. We introduced Findel – a declarative financial DSL built upon ideas from previous research in financial engineering. Formalizing contract clauses using Findel makes them unambiguous and machine-readable. We proved Ethereum to be a suitable platform for trading and executing Findel contracts.

Nevertheless, the whole smart contract field is still in its infancy. Programmers who wish to implement a usable smart contract for handling financial agreements need to be aware of the forthcoming challenges: from fundamental limitations of the blockchain network architecture to imperfect development environment.

[17] Szabo makes a case against state variables in general, stating that "they should be avoided unless utterly necessary".

References

ABC16. Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts. *IACR Cryptology ePrint Archive*, 2016:1007, 2016.

Bis16. Statistical release. otc derivatives statistics at end-june 2016, 2016. `https://www.bis.org/publ/otc_hy1611.pdf`.

Bro16. Browser-solidity online compiler, 2016. `https://ethereum.github.io/browser-solidity/`.

But14. A next-generation smart contract and decentralized application platform, 2014. `https://github.com/ethereum/wiki/wiki/White-Paper`.

CBB16. Christopher D. Clack, Vikram A. Bakshi, and Lee Braine. Smart contract templates: foundations, design landscape and research directions. *CoRR*, abs/1608.00771, 2016.

CF14. Sophia Yakoubov Conner Fromknecht, Dragos Velicanu. A decentralized public key infrastructure with identity retention. Cryptology ePrint Archive, Report 2014/803, 2014. `http://eprint.iacr.org/2014/803`.

Cha15. Shahbaz Chaudhary. Adventures in financial and software engineering, 2015. `https://falconair.github.io/2015/01/30/composingcontracts.html`.

CRR79. John C Cox, Stephen A Ross, and Mark Rubinstein. Option pricing: A simplified approach. *Journal of financial Economics*, 7(3):229–263, 1979.

dC16. Michael del Castillo. Jp morgan, credit suisse among 8 in latest bank blockchain test, 2016. `http://www.coindesk.com/jp-morgan-credit-suisse-among-8-in-latest-bank-blockchain-test/`.

Eth17. Ethstats, 2017. `https://ethstats.net/`.

FpE17. Functional programming for ethereum, 2017. `https://github.com/fp-ethereum/fp-ethereum`.

FSNB09. Simon Frankau, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19(01):27–45, 2009.

Gai11. Jean-Marie Gaillourdet. A software language approach to derivative contracts in finance. 2011. `http://ceur-ws.org/Vol-750/yrs06.pdf`.

Gib13. Jeremy Gibbons. Functional programming for domain-specific languages. In *CEFP*, volume 8606 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2013.

Gre16. Gideon Greenspan. Why many smart contract use cases are simply impossible, 2016. `http://www.coindesk.com/three-smart-contract-misconceptions/`.

Hvi10. Tom Hvitved. A survey of formal languages for contracts. In *Fourth Workshop on Formal Languages and Analysis of Contract–Oriented Software (FLACOS10)*, pages 29–32. Citeseer, 2010.

Ico17. Icos, token sales, crowdsales, 2017. `https://www.smithandcrown.com/icos/`.

JE03. Simon L. Peyton Jones and Jean-Marc Eber. How to write a financial contract. *The Fun of Programming*, 2003.

JES00. Simon L. Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering, functional pearl. In *ICFP*, pages 280–292. ACM, 2000.

Kho16. Dmitry Khovratovich. debt.sol, 2016. `https://gist.github.com/khovratovich/45f68082b556b45eb64e8e1c3eb82892`.

LC16. Karen Lewison and Francisco Corella. Backing rich credentials with a blockchain pki, 2016. https://pomcor.com/techreports/BlockchainPKI.pdf.
Lex00. Ocaml at lexifi, 2000. https://www.lexifi.com/blogs/ocaml.
Mor16. Sofus Mortensen. Universal contracts, 2016. https://github.com/corda/corda/tree/master/experimental/src.
Ora16. Oraclize, 2016. http://www.oraclize.it/.
Rea16. Reality keys, 2016. https://www.realitykeys.com/.
Rev16. Raine Rupert Revere. What is the difference between transaction cost and execution cost in browser solidity?, 2016. https://ethereum.stackexchange.com/q/5812/5113.
Sch13. Todd Schiller. Financial DSL listing, 2013. http://www.dslfin.org/resources.html.
Sch14. Steffen Schuldenzucker. Decomposing contracts. 2014. http://www.ifi.uzh.ch/ce/people/schuldenzucker/decomposingcontracts.pdf.
Sch16. Steffen Schuldenzucker. An axiomatic framework for no-arbitrage relationships in financial derivatives markets. 2016. http://www.ifi.uzh.ch/ce/publications/LPT.pdf.
Sir16. Emin Gün Sirer. Thoughts on the dao hack, 2016. http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/.
STM16. Pablo Lamela Seijas, Simon Thompson, and Darryl McAdams. Scripting smart contracts for distributed ledger technology. Cryptology ePrint Archive, Report 2016/1156, 2016. http://eprint.iacr.org/2016/1156.
Sza97. Nick Szabo. Formalizing and securing relationships on public networks, 1997. http://journals.uic.edu/ojs/index.php/fm/article/view/548.
Sza02. Nick Szabo. A formal language for analyzing contracts, 2002. http://nakamotoinstitute.org/contract-language/.
Tls16. Tlsnotary, 2016. https://tlsnotary.org/.
Tok16. Ethereum improvement proposal: Token standard, 2016. https://github.com/ethereum/EIPs/issues/20.
vS07. Anton van Straaten. Composing contracts, 2007. https://web.archive.org/web/20130814194431/http://contracts.scheming.org.
Wal12. Channing Walton. Scala contracts project, 2012. https://github.com/channingwalton/scala-contracts/wiki.
Woo14. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. http://gavwood.com/paper.pdf.
Wor16. Gross domestic product 2015, 2016. http://databank.worldbank.org/data/download/GDP.pdf.
Wor17. Worldcoinindex, 2017. https://www.worldcoinindex.com/coin/ethereum.

A Examples

– A **fixed-rate currency exchange**: the owner sells €10 for $11.

$$\mathrm{And}(\mathrm{Give}(\mathrm{Scale}(10, \mathrm{One}(EUR))), \mathrm{Scale}(11, \mathrm{One}(USD))$$

– A **market-rate currency exchange**: the owner sells €10 at market rate as reported by the gateway at *addr*.

$$\mathrm{Scale}(10, \mathrm{And}(\mathrm{Give}(\mathrm{One}(EUR)), \mathit{ScaleObs}(\mathit{addr}, \mathrm{One}(USD))))$$

– A **zero-coupon bond**: the owner receives \$100 at t_0.

$$\text{Timebound}(t_0 - \delta, t_0 + \delta, \text{Scale}(100, \text{One}(USD)))$$

– A **bond with coupons**: the owner receives \$1000 (face value) in three years (maturity date) and two coupon payments of \$50 at regular intervals before the maturity date.

$$\text{And}(\text{At}(\text{now} + 3 \text{ years}, c_{face}), \text{And}(\text{At}(\text{now} + 1 \text{ years}, c_{cpn}), \text{At}(\text{now} + 1 \text{ years}, c_{cpn})))$$

where

$$c_{face} = \text{Scale}(1000, \text{One}(USD)), \quad c_{cpn} = \text{Scale}(50, \text{One}(USD))$$

– A **future** (a **forward**[18]): parties agree to execute the underlying contract c at t_0.

$$\text{Timebound}(t_0 - \delta, t_0 + \delta, c)$$

– An **option**: the owner can choose at (European option) or before (American option) time t_0 whether to execute the underlying contract c.

$$\text{Timebound}(t_0 - \delta, t_0 + \delta, \text{Or}(c, \text{Zero}))$$

$$\text{Timebound}(now, t_0 + \delta, \text{Or}(c, \text{Zero}))$$

– A **binary option**: the owner receives \$10 if a predefined event took place at t_0 and nothing otherwise.

$$\text{If}(addr, \text{Scale}(10, \text{One}(USD)), \text{Zero})$$

[18] In traditional finance, a future is a standardized contract while a forward is not. This distinction is not relevant for our model.