Distributed consensus for models of logic

Daniel Murfet

June 6, 2019

Abstract

Using proof-of-work as a distributed consensus algorithm we propose a practical method for implementing a network which maintains a distributed and evolving model of an arbitrary first-order logical theory, in a way compatible with a certain class of changes in the underlying theory. The network which maintains the model also maintains a list of conjectures about the model, the state of knowledge about which is encoded in a prediction market.

1 Introduction

Distributed consensus is the cryptographic process that members of a cryptocurrency network use to synchronise their individual copies of the complete history of transactions. This globally coherent history is usually referred to as the *blockchain* [2]. The fact of this global coherence, together with the verification of the lack of double-spending at the time of incorporation of transactions into the blockchain, has an emergent consequence: the existence of a unit of cryptocurrency (Bitcoin, say) with certain logical properties.

In this sense the Bitcoin network is a *logical community*: the energy expended in the proof-of-work protocol can be viewed as the cost of instantiating an emergent logical order with desirable properties (scarcity) on top of more fundamental computational degrees of freedom (the bits of the computers in the network) which do not possess those properties.[1] This note explores the general case of this idea, where the logical order is described by an arbitrary first-order theory, and the analogue of the blockchain maintains a distributed globally coherent *model* of that theory, which may be modified locally by any participant in the network according to a prescribed set of incremental transformations. We call this globally coherent history of transformations the *proofchain*. We hope that proofchains can help to enable new forms of cooperation, by creating logical communities with economic incentives that are more productive than the prevailing ones.

[1] Much as models of topological order based on surface codes [10] expend considerable time and classical computation in order to engineer emergent topological order on top of lattices of qubits.

The details of the proposal are given in the next section. To motivate the definitions, let us first imagine a decentralised community maintaining a shared repository of formally verified software components. To explain how novel kinds of markets could help incentivise the activity of such communities, we provide the following cartoon:

Example 1.1. Suppose a user wishes to commission software with specification T consisting of minimal requirements S plus security constraints S', so that $T = S + S'$. This request is posted to the proofchain network together with cryptocurrency bounties funding the work, and the following activity ensues:

- Users survey the existing library $\{(T_i, M_i)\}_{i \in I}$ of software maintained in a tamper-evident and distributed way by the proofchain. The library consists of specifications T_i together with software M_i fulfilling the specification. If necessary new components are commissioned with their own bounties. Once the work is complete and code M satisfying S has been synthesised, users receive the bounty associated to S.

- To determine whether M satisfies the security constraints S' a prediction market is opened, capitalised by the bounty, where users buy and sell shares in the payoff from a successful proof of the conjectures S' about the code M. We refer to this as a *conjecture market*. The price of these shares gives an objective measure of the knowledge of the community about whether M satisfies S' [15].

  A user who possesses a proof π of S' can buy all open shares in the market, and then broadcast π onto the network, at which point the network protocol automatically closes the market and distributes the payoffs. To produce a proof π means, generally speaking, to first prove properties $\psi_1, \dots, \psi_k$ of the components $N_1, \dots, N_k$ used to assemble M. The market value of π determined by the bounty implies a market value for $\psi_1, \dots, \psi_k$, by arbitrage between conjecture markets, in a recursive process. Users react to the market incentive by proving the fine-grained conjectures, and the results are collected into a proof π of S' for M.

The integrity of this process is guaranteed by the distributed consensus mechanism.

2 First-order logic

2.1 Languages, theories and models

Our main reference is [1, §X.1]. A *first-order language* L is given by a collection of "sorts" (or "types") $X, Y, \dots$ collections of relation symbols $R, S, \dots$ and of function symbols $f, g, \dots$ and possibly some constants $c, d, \dots$. The relation symbols may include properties (unary relations). Each relation symbol is given together with the sorts of its arguments. For instance, R could be a binary relation taking an argument x of sort X and an argument y of sort Y, in which case we write $R \subseteq X \times Y$. Similarly each function symbol f of the language is given with the sorts of its arguments, and the sort of its output. We write

$$f : X_1 \times \cdots \times X_n \longrightarrow Y$$

if f takes n arguments of sorts $X_1, \ldots, X_n$ respectively to a value of sort Y. Each constant c of the language is given with a specified sort X, and we write $c \in X$. We also assume that for each sort X the language has infinitely many variables $x_1, x_2, x_3, \ldots$ of that sort. With such a language L one can build up terms and formulas in the usual way, and define an *interpretation* of the language in sets or in an arbitrary topos [1, §X.2]. A first-order *theory* T in the language L is just a set of formulas, then called the *axioms* of T. A *model* of T is an interpretation M of L in which all the axioms of T are valid.

Remark 2.1. For the application we have in mind, the relevant kind of topoi (besides the topos of sets **Sets** and of finite sets **FinSets**) are those obtained as the category of types and terms in a higher-order logic; see [4, §II.1] and [3, Lecture 9].

Example 2.2. The theory of *abelian groups* can be formulated in the language L containing one sort X, no relation symbols, two function symbols $+ : X \times X \longrightarrow X$ and $- : X \longrightarrow X$ and one constant $0 \in X$, with axioms including $(x + y) + z = x + (y + z)$. A model of this theory in the topos of sets is simply an abelian group, in the usual sense.

Example 2.3. The theory of *standard Bitcoin* can be formulated in the language L containing two sorts C, P (representing the set of coins and public keys, respectively) and a single function $f : C \longrightarrow P$. A model M in **Sets** is just a function assigning public keys to coins. Such a function represents the state of the allocation of cryptocurrency at any given time.[2] Then, roughly speaking, the Bitcoin blockchain is a sequence

$$M_1, M_2, M_3, \ldots, M_n, M_{n+1}, \ldots$$

of models where M_{n+1} is obtained from M_n by an allowed operation (a transaction).

3 The proofchain

Throughout this section we fix a topos $\mathcal{E}$ where we take models. We assume that there is an effective computational means of describing models in $\mathcal{E}$ by binary strings (for example, models in the topos of finite sets have such a description, as do models in the topos of types and terms in a higher-order logic as in [4, §II.1], so this is not a strong restriction). We call such strings *codes* and write $c(M)$ for the code of a model M in $\mathcal{E}$.

Definition 3.1. A *logical order* is the data of

[2] In Bitcoin coins are defined to be chains of transactions starting from either the genesis block or blocks where the coins were "mined" but given that the blockchain is a linear chain of hashes and there are ultimately only finitely many satoshis, there is effectively an enumeration of satoshis according to which we may take $C = \{1, \ldots, N\}$ for some large integer N.

- A first-order geometric theory T,
- A finite set of *conjectures*, which are formulas in the language of T, and which are associated to particular versions of the model identified by transaction identifier,
- A set of *allowed transactions* on models of T.

The allowed transactions are represented by a polynomial time boolean-valued computable function $A(p, c(M), c(M'), \tau)$ of a public key p, a pair M, M' of models represented by their codes, and an algorithm τ which represents the change, that is, which computes M' from M. The function A evaluates to true if and only if the user with public key p is allowed to transition the model from M to M' via the transaction τ. For more details on the nature of the algorithms τ that we allow as transactions, see Section 5.3.

We restrict to geometric theories [1, §X.3] so that we have a good theory of "transfer" of models along change of logical order (see Section 5.2 and Section A). This may not be necessary. The restriction to geometric theories is a real restriction, but Vickers has argued for why it is reasonable in the setting of software specification [5, 6].

In the remainder of this section we give a preliminary sketch of the proofchain protocol, with minimal detail about the role of conjectures or the language in which the transactions τ are to be written. For more information see Section 5 below.

Example 3.2. In the case of $T_{Bitcoin}$ the function $A(p, c(M), c(M'), \tau)$ would evaluate to true if and only if the function $f' : C \longrightarrow P$ giving the data of M' is obtained from the function $f : C \longrightarrow P$ giving the data of M by replacing a pair (c, p_s) consisting of a single coin c and a public key p_s (the source) with a pair (c, p_t) (the target), $p = p_s$ and τ is the algorithm which performs that rewrite of f to f'.

Example 3.3. Let T be the theory of linear orders in the sense of [1, §VIII.8] so that the category of models $\mathrm{Mod}(T)$ in the topos **Sets** is equivalent to the category of simplicial sets [1, Theorem VIII.5]. Let the set of allowed transactions be the addition or deletion of a single vertex in the model M, that is, in the simplicial set encoded by the model, together with the necessary modifications to the face and degeneracy maps so that the constraints of being a simplicial set remain satisfied. Moreover by using the input p the function A could effectively allow public keys to *own* subsets of vertices, in the sense that only a given user is allowed to initiate transactions that remove those vertices.

The proofchain, specified below, solves the problem of distributed and decentralised consensus about a single model M of the theory T, whose evolution in time is constrained by the set of allowed transactions described by A. That is, the proofchain instantiates the given logical order. The details are the obvious modifications of [2]. The proofchain is, roughly speaking, a crytographically secured sequence

$$c(M_1), (p_1, c(M_1), c(M_2), \tau_1), c(M_2), (p_2, c(M_2), c(M_3), \tau_2), \ldots, c(M_n) \tag{1}$$

where M_i are models for $i \geq 1$, the p_i are public keys, and for all $i \geq 1$

$$A(p_i, c(M_i), c(M_{i+1}), \tau_i) = \text{true}\,.$$

In order that the ith transaction is authenticated as being initiated by the user with public key p_i, and in order that this user cannot deny having sent the transaction, we require as in the Bitcoin blockchain that the user digitally signs the pair

$$c(M_i), (p_i, c(M_i), c(M_{i+1}), \tau_i)$$

with their private key. A *block* consists of a sequence (1) satisfying the given constraints, together with the digital signatures and the hash of another block (the parent block) and a nonce. The proof-of-work consists of incrementing the nonce until a value is found that gives the block's hash some required number of zero bits.

The steps to run the network are as follows [2, §5]:

(1) New transactions are broadcast to all nodes

(2) Each node collects new transactions into a block

(3) Each node works on finding a difficult proof-of-work for its block

(4) When a node finds a proof-of-work, it broadcasts the block to all nodes

(5) Nodes accept the block only if all transactions in it are valid

(6) Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash

The only difference to the Bitcoin blockchain protocol lies in step (5) where we use a more general notion of validity, based on the function A. Nodes always consider the longest chain to be the correct one and will keep working on extending it, and in this way the energy expended in proof-of-work creates a distributed consensus on a model M. The other details (such as the block verification protocol) follow Bitcoin.

Summary: the proofchain secures the fact that the data M is (in the same approximate sense as in the usual blockchain) guaranteed to both be a model of the given logical theory, and also to have evolved from its initial state according to the allowed set of possible transactions as described by the logical order.

Remark 3.4. The computational cost of verifying the constraint on transactions in the Bitcoin blockchain (of no double spending) is low and does not scale with the number of coins or available public keys. In principle, however, our proofchain has an arbitrarily high computational cost per transaction: the cost of evaluating the function A must be paid both by nodes forming blocks, and also in the block verification stage by other nodes before they accept and retransmit blocks. This cost involves verifying the axioms of the theory T for the new state of the model, and these axioms may involve quantification over a large set (say of vertices in a simplicial set, in the case of the theory of linear orders).

On the one hand this cost can be compensated by either mining (as in Bitcoin) or by transaction fees, and on the other hand one should design the set of allowed transactions so that A has a time complexity which is as low as possible. Ideally, it should be possible to infer cheaply from the hypothesis that M is a model and that τ belongs to an allowed set of transactions that M' is also a model *without directly checking* that M' is a model (i.e. independent of this knowledge of how it was constructed). For more see Section 5.4.

4 Examples

Just as proof-of-work in the Bitcoin network secures the existence of Bitcoins that can be exchanged between network participants, proof-of-work in our setting could for example secure the decentralised existence of a finite simplicial set (which is a model of the theory of linear orders) to which network participants can add and destroy vertices, edges and faces. Perhaps more usefully: a user who wishes a program authored to some specification could encode the specification as a first-order theory and then initialise a proofchain (incentivised by cryptocurrency) implementing the logical order described by that theory: the analogue of transactions on this proofchain would be formally verified steps towards the implementation of the specification, so that users can collaborate in a decentralised and trustless way in the authoring of a complicated codebase; see Section 4.1.

First, we make a general comment. Let (T, A) be a logical order in the sense of the previous section. It is good to keep in mind that the computational cost of instantiating this logical order, via the proofchain, means that there must be a strong rationale (economic, aesthetic or otherwise) for the existence of this logical order. Ultimately this means that there must be value in

(a) every member of the network having the *same* data M, and in

(b) every member of the network being confident that the data M does in fact obey the axioms of T, that is, that M is a *model* of the logical theory.

In the case of $T = T_{Bitcoin}$ the economic rationale is clear: obviously a currency is only valuable if (a) everybody agrees on who owns what, and (b) if the currency is scarce, in the sense that it cannot be double-spent.

4.1 Trustworthy software

Models of software are often formalised logically as a set of entities (sorts) and functions or relations between these entities, subject to some contraints (axioms). This *specification* is then given to a software engineer whose job it is to *implement* the specification, which means to write a program which implements the specified behaviour. The problem here is obvious: how do you *trust* that the program in fact implements the specification?

One of the popular specification languages is Z [7] which is based on first-order logic and Zermelo-Fraenkel set theory. It is an international standard widely used in industry. From the point of view of this note a specification is a first-order theory T and an implementation of the specification is a model M of T in a topos $\mathcal{E}$ which is the syntactic category of the implementation programming language [4]. From this point of view the natural class of implementation languages are functional languages, obtained as extensions fo the pure simply-typed lambda calculus, but see Section 5.3.

One of the central problems of modern software engineering is *trust*: how can the author of the specification be sure that the software engineer has delivered an implementation that actually obeys the constraints of the specification? In most parts of software engineering there is no practical cost-effective solution to this problem and the guarantee is merely a social contract (if your code has a bug you will be fired). In industrial and military settings there are sometimes enough resources that the programmer or a third party will be paid to prepare a *proof* of correctness, which is then presented to the relevant stakeholders; an important modern example is seL4 [8]. But this proof only guarantees correctness of a particular current state of the model, and may be worthless as soon as the model is changed [9], see Section 5.2 below.

In practice the authority of the proof claim of the verification agent is also partly dependent on the reputation of the agent, because the party who generates the specification may not have the necessary understanding to fully check a proof that a given implementation is correct. This problem becomes even more serious when a *third party*[3] wishes to assess the reliability of the correctness claim of the implementation of the specification. Let us suppose that some set of third parties wishes **both** to have high confidence in the correctness of an implementation without relying on the authority of a central party **and** further these parties wish to collaboratively extend the implementation over time.

Under these conditions there is economic value in

(a) every member of the network having the *same* implementation M, and

(b) in every member being confident that the implementation M is correct

and so this is valid use-case for a proofchain. In summary:

Example 4.1 (Specification). The first-order theory T is the specification of the desired software system. The set of allowed transactions A are an allowed set of atomic changes to the codebase. In its initial state the proofchain is a "trivial" solution to the constraints, which is then extended by allowed transactions to a nontrivial solution. Some thought is required to allow for "partial" specifications which are compatible with this kind of approach based on extensions from a trivial solution (but this is a standard research topic in specification languages and trustworthy computing).

Since each transaction may be attributed to a public key, in addition to transaction fees and mining fees for rewarding nodes in the network, it is possible to incentivise the extension of the implementation by cryptocurrency bounties to individual programmers.

[3] By which we mean a party which is not necessarily the originator of the specification nor the author of the proposed implementation.

5 Further details

As mentioned in the introduction, proofchains may enable new forms of cooperation, by creating logical communities with strong economic incentives. In order that the resulting patterns of cooperation are stable across time, we view the following problems as fundamental obstacles that a successful design must overcome:

- **Stability to changes in the logical order.** It is an empirical fact that successful long term patterns of cooperation involve institutions which evolve (in a controlled way) with time. We therefore introduce the notion of *change of logical order* (Section 5.2) and the transformation of the proofchain under such changes *Examples*: changing specification for a programming project (Section 4.1), modularity and re-use of programs in large projects (Section 5.2.1).

- **Stability to uncertainty.** A model of a first-order theory is a mathematical object which involves certainty about properties of the data in the model. However, in practice there are useful intermediate degrees of belief, and to avoid discontinuous barriers to entry it seems important to capture these degrees of belief in a market mechanism. We therefore introduce *conjecture markets* (Section 5.1).

5.1 Conjecture markets

A logical order consists of a first-order language together some formulas called *axioms* and some formulas called *conjectures*. The proofchain protocol verifies that the data of the model satisfies the axioms at all times. However, the conjectures are "soft" constraints in the sense that they are not required to hold. Instead, the state of belief about each conjecture is encoded in a *prediction market* which is maintained as part of the proofchain protocol. Prediction markets are a simple and elegant distributed method of incentivising the sharing of information [15]. Given a proposition P, the market trades in shares of a fixed payoff in the event that P is proven to be true. The more certain the market is that P is true, the higher the price of each share will be to acquire.[4]

Ideally, a user of the network can query this prediction market to obtain an estimation of the likelihood of the truth of conjecture which accurately represents the state of the knowledge of the participants in the network. The key to the prediction market reflecting the true state of knowledge of the participants is that each user is incentivised economically to contribute whatever knowledge they possess about the conjecture, and indeed, that users are incentivised to invest whatever computational or cognitive resources are required to *determine* the truth of the conjecture.

It is desirable that there is no hard boundary between axioms and conjectures: instead, there should be a mechanism in the protocol for conjectures to be converted into axioms,

[4] Parties wishing to incentivise the resolution of a conjecture can donate funds to increase the payoff.

when a proof (or disproof) of the conjecture is found by the market mechanism. In this way an initially modest set of axioms can evolve into a stronger theory over time.[5]

The problem with conjecture markets. A conjecture is a formula ψ that is either true or false for any given model M of the theory T. There are various propositions derived from a conjecture ψ that could be the objects of a prediction market, for example:

- A proof of ψ will be found at some future time for the current model at that time.
- A proof of ψ will be produced for a particular model already on-chain.

In both cases the payoff is triggered when a proof is broadcast over the network. While the second kind of conjecture has many complexities, the first kind of conjecture suffers from a fatal flaw: why pay for someone to prove properties of a model you haven't seen? There could exist a model which trivially satisfies the constraints, or satisfies the constraints but has other undesirable properties. For this reason we focus in this document on the second kind of conjecture, about known versions of the model.

The primary problem with such conjectures is *fragmentation* of the market and the consequent *illiquidity*. If a conjecture ψ is posed about model M, and then the proofchain evolves to a new state of the model M', the truth of ψ for M, M' may be logically independent. In general the proofchain may contain thousands of *a priori* unrelated conjecture markets, all involving the same formula ψ but different versions of the model.

Market liquidity. We hope to solve the liquidity problem in conjecture markets with a combination of market making algorithms provided by the platform itself, and trader logicians who make money by arbitrage between markets. In the limit of infinitely many well-capitalised traders interested in each state of the model, the second source of liquidity will be sufficient. However, this is not a realistic assumption, especially at the beginning. Automatic market making algorithms can make prices consistent across many conjecture markets which involve trivial differences of the model, and with more research this role can be extended gradually to handle less trivial differences. However, such market makers will have to be carefully designed to mitigate the risk of large losses.

Here is an example of market making performed by human traders. In the context of Example 1.1, suppose a trader figures out a proof ζ of the conjecture π on the hypothesis of propositions $\psi_1, \dots, \psi_k$ about sub-components $N_1, \dots, N_k$ of the program M, but they do not yet know how to prove these propositions. They keep to themselves the proof ζ of

$$\big(\psi_1 \wedge \cdots \wedge \psi_k\big) \Longrightarrow \pi$$

while fronting their own funds to establish conjecture markets ψ_i with bounties $c_1, \dots, c_k$. Other users see the open conjectures about the ψ_i and provide proofs. The original trader collects these proofs, combines them with ζ, and broadcasts the resulting proof π of S' in

[5] This potentially one way to avoid the problem of a high barrier of entry to the authoring of formally verified software.

order to claim the bounty $b_{S'}$. As a result of this activity, they make a profit $b_{S'} - \sum_i c_i$. This reward does not come without risk: the conjectures ψ_i may be *false*, or another user may know ζ and claim the bounty $b_{S'}$ without having to pay for the proofs of the ψ_i. This risk-taking is a form of market making that hopefully works to spread liquidity from highly capitalised conjecture markets (valuable properties of important programs) to other markets, in a recursive fashion.

Automatic market makers. It turns out that similar problems have been studied in the special case of *propositional* logic under the name *combinatorial prediction markets*, where Hanson's logarithmic market scoring rule (LSMR) [19] provides a market maker with bounded losses. However LSMR pricing is #P-hard, and so in practice one needs to be use some kind of approximation; see also [18, 17, 20]. From [17] we quote:

> Operating a combinatorial exchange, beyond the daunting computational challenge [Fortnow et al. 2004], is impractical for a more pedestrian reason: it is hard to imagine how any given trade, selected from the unimaginably large sea of choices, would happen to have a matching counter trade waiting to execute against it. For this reason, an automated market maker, which always offers a price for any security no matter how complex, is better suited for combinatorial prediction markets.

This is partly a machine learning problem related to probabilistic relaxations of inference algorithms; variational inference in the context of graphical models are mentioned in [17]. This seems related to the proposal in [21].

Open problem 5.1. Design low computational complexity market makers for conjecture markets in higher-order logic with some kind of approximate bounds on losses.

Initially conjecture markets should be as simple as possible, with easily understandable market making algorithms. Over time the tools and trading strategies of users will evolve to become more complex, as the markets mature. In defense of the idea, it is worth noting that the alternatives to conjecture markets also have liquidity close to zero (where can you currently go and buy a proof?). At least the above proposal gives a plausible mechanism for getting such markets off the ground.

5.2 Change of logical order

As explained in the introduction, we view the energy expended by the proof-of-work protocol in a proofchain as instantiating an emergent logical order. We adopt as a fundamental aspect of our design the possibility of *changes* in this logical order.

For example, during a large software engineering project it is natural that the initial specification will evolve. If such evolution is forbidden, the probability of project failure is increased and the initial cost and difficulty of authoring the specification is also increased. Some other examples of desirable changes in a logical order include:

- The conversion of proven/disproven conjectures to axioms.
- Re-use and modularity (see Section 5.2.1)
- Blockchain governance

The economic value in a proofchain lies in the trust that the users of the network have in the fact that they share the *same* model data, and that this model is both *correct* (in the sense that it is truly a model of the theory) and that it was constructed from an initial state by a *sequence of allowed transactions.* The knowledge about the model encoded in the conjecture market also has economic value. It is important that changes in the logical order preserve this economic value.

We therefore require a notion change of logical order which is both weak enough that it contains the natural examples, but strong enough that we can effectively transfer models and their associated conjecture markets along such changes. Fortunately topos theory provides a natural solution: it is always possible to transfer models along *geometric* morphisms of topoi [1, Corollary §X.6]. Associated to a first-order geometric theory T is the *classifying topos* $\mathcal{B}(T)$ which itself has a purely syntactic description [11, §D1.4], [3, Lecture 15], see also [1, §X.5]. The *geometric morphisms* [1, Ch. VII] provide the correct notion of morphisms between topoi and the universal property of the classifying topos [1, §VIII.3] is a natural equivalence of categories

$$\underline{\mathrm{Hom}}(\mathcal{E}, \mathcal{B}(T)) \cong \mathrm{Mod}(T, \mathcal{E})$$

for any cocomplete topos $\mathcal{E}$, where $\underline{\mathrm{Hom}}$ denotes the category of geometric morphisms and natural transformations. A geometric morphism $f^* : \mathcal{B}(T) \longrightarrow \mathcal{B}(T')$ induces a functor

$$f^* : \mathrm{Mod}(T, \mathcal{E}) \longrightarrow \mathrm{Mod}(T', \mathcal{E})$$

between the categories of models of T and T' in a topos $\mathcal{E}$. By the universal property of classifying topoi, a geometric morphism $\mathcal{B}(T) \longrightarrow \mathcal{B}(T')$ is the same data (up to canonical isomorphism) as a model M of the theory T in the topos $\mathcal{B}(T')$.

Definition 5.2. A *change of logical order* $T \longrightarrow T'$ is a geometric morphism

$$f^* : \mathcal{B}(T) \longrightarrow \mathcal{B}(T')$$

for which the associated model M of T in $\mathcal{B}(T')$ is of finite type. Moreover, we require that the induced functor on models is compatible with the predicates A, A' of the logical orders T, T' in the following sense: for any public key p and models M, M' of T, and term τ in the transaction language, we have

$$A(p, c(M), c(M'), \tau) \Longrightarrow A'(p, c(f^*M), c(f^*M'), \tau)\,. \tag{2}$$

As a consequence of the condition (2) we may apply a change of logical order to the proofchain for a logical order (T, A) to generate a proofchain for the logical order (T', A').

Open problem 5.3. How should changes of logical order interact with conjecture markets? A geometric morphism preserves the truth of propositions about the model, but there will be examples where it is *easier* to guess the truth of a conjecture in the image (for example if the geometric morphism is a degenerate translation) and so one should not expect the market price of a conjecture to be stable across a change of logical order. We hope that for some changes of logical order a simple market making algorithm can establish consistent prices, and in the remaining cases we rely on human traders to perform the market making.

6 Further directions

6.1 Networks of proofchains

Suppose a first-order geometric theory T represents the specification of a piece of software. The proofchain is a distributed model of this theory, that is, an implementation of the specification, in a particular target language represented by the choice of topos $\mathcal{E}$. Given that software is generally constructed in a hierarchical fashion, the most natural way to author such an implementation is to re-use components that have been previously specified and implemented. This requires interactions between proofchains, and should be based on the 2-category of Grothendieck topoi and its 2-limits and colimits.

For example, a pair of specifications T, T' containing a common subset S is associated to a pair of changes of logical order $S \longrightarrow T, S \longrightarrow T'$ and the union of the specifications along this subset has for its classifying topos the pushout of $\mathcal{B}(T), \mathcal{B}(T')$ along $\mathcal{B}(S)$. It is therefore reasonable to imagine a higher-level network of proofchains, transitions between which are mediated by software-encoded geometric morphisms.

To design a network of proofchains the primary question seems to be how to interlink their conjecture markets. If the market prices a conjecture C about a component M at a certain value, how should this influence the price of a conjecture C' about an implementation which uses the component M? In some cases this is trivial: if C' is logically entailed by C *and everyone knows the proof*, the price of C' should be higher than the price of C (assuming the fixed payout is the same for both conjectures). However, if the truth of C' also involves a series of other conjectures about other components, each with their own market price, it is unclear how to rigorously determine an optimum price. However, if we design the markets correctly, we should be able to rely on *arbitrage across conjecture markets* to determine these prices dynamically; see Section 5.1.

This arbitrage will serve to price mathematical knowledge, since if the market does not know that C' is entailed by C, any user who possesses a proof can make a successful trade. In this way a user of the network who proves a conjecture C about a fundamental component, can use that knowledge to trade against the markets for conjectures $C'_1, C'_2, \ldots$ about higher-level implementations. This provides an economic model for some kinds of mathematical activity which benefit the users of the network. This proposal is a more finely grained version of some of Hanson's suggested reforms of science [13, 14].

6.2 Implementation language

The models in a given proofchain are all in a fixed topos $\mathcal{E}$, which represents the *target language* of the models. For simple combinatorial models the natural topos is **FinSets**. However, for examples where the implementation consists of a piece of software, the most natural examples are the topoi associated to type theories with natural numbers objects, which while not Turing complete are arguably sufficiently expressive for practical programming [4, §III.3]. Recall that there is a correspondence between simply-typed lambda calculi and Cartesian closed categories [4, Part I] and similarly between intuitionistic type theories (also called higher-order logics) and topoi [4, Part II]. Roughly speaking, an intuitionstic type theory is a simply-typed lambda calculus together with facilities for reasoning about properties of programs; for an introduction see [3, Lecture 9].

For many problems the appropriate language for implementation will be finite-state machines, Turing machines, instruction sets of CPU architectures, or even imperative programming languages such as C, etc., but many of these have well-understood translations into lambda calculi; see [16].

6.3 Transactions

The definition of a logical order involves a predicate A which determines the set of allowed transactions. The design of this predicate is by far the most intricate part of constructing a logical order, since

- A change $T \longrightarrow T'$ in logical order must be compatible with the predicates A, A'.
- In order for conjecture markets to function across time, most conjectures should be stable with respect to most changes in the model allowed by the predicate A.

I do not currently understand how to design transactions. However, here are some observations. We restrict to the examples of $\mathcal{E} = \mathbf{FinSets}$ and $\mathcal{E} = \mathcal{C}(\mathbb{T})$ the topos associated to a intuitionstic type theory $\mathbb{T}$ as in Lambek-Scott [4, §II.12].

Example 6.1. In the case of $T = T_{Bitcoin}$ we take $\mathcal{E} = \mathbf{FinSets}$ and the allowed transactions change a pair (c, p_s) in the graph of f to a pair (c, p_t). Neither changes in logical order nor conjecture markets seem particularly relevant, but it is worth noting that the set of allowed changes is "local" in the sense that it affects only the value of the function a single coin.

In the case $\mathcal{E} = \mathcal{C}(\mathbb{T})$ the sorts $X, Y, \ldots$ relations $R, S, \ldots$ functions $f, g, \ldots$ and constants $c, d, \ldots$ of the theory are interpreted by objects, subobjects of Cartesian products and morphisms of $\mathcal{E}$ respectively. However, these structures are in turn represented by closed terms in the language $\mathbb{T}$. This is explained in some detail in [3, Lecture 15]. One should think of $\mathbb{T}$ as a kind of set theory and, just as in ordinary set theory, "sets" are actually closed terms in a certain language. For example, the objects of $\mathcal{E}$ are equivalence classes of closed terms $\alpha : PA$ for some type A of $\mathbb{T}$ where P stands for the power set operation, and morphisms are also equivalence classes of closed terms.

Suppose the theory consists of a single sort X and function symbol f, together with an axiom $f \circ f = f$. Let M be the model which interprets the sort X as $[\alpha] \in \mathcal{E}$ with closed term $\alpha : PA$, so that the function symbol $f : X \longrightarrow X$ must be interpreted by an equivalence class $[F]$ for a closed term $F : P(A \times A)$ which is provably functional. Moreover, for this to be a model of the theory, we must have provability of

$$\vdash F \circ F = F$$

in the type theory $\mathbb{T}$, where $\circ$ here denotes the usual construction of composition of graphs in set theory. Let M' be another model with data (α', F'). To describe an allowed set of transactions of the form

$$\tau : (\alpha, F) \dashrightarrow (\alpha', F')$$

is to describe an allowed set of transformations of terms $\alpha \dashrightarrow \alpha', F \dashrightarrow F'$ in the language of $\mathbb{T}$. We leave the precise description of such allowed kinds of transformations for later, but ideally each allowed transaction τ would come with a proof in $\mathbb{T}$ of the sequent

$$F \circ F = F \vdash F' \circ F' = F' \, .$$

In this case the computational cost to miners of verifying transactions on the proofchain should be reasonable (see Remark 3.4). Moreover, using the internal language of topoi and its compatibility with geometric morphisms, the problem of understanding which geometric morphisms are compatible with a given set of transactions (specified in this way as transformations of terms) seems like a tractable problem.

Let us address the second concern, about conjecture markets. Given an allowed transaction τ we define the *support* of τ to be the symbols in the model (i.e. sorts, relations, functions and constants) whose interpreting term in the language of $\mathbb{T}$ is modified by τ. A conjecture C in the theory which does not involve any of the symbols in the support of τ is true for M if and only if it is true for M', and so its market price should remain unchanged under the transaction τ. It is possible by inspecting τ to say more on a case-by-case basis, but again this is probably a role best left to parties performing arbitrage on conjecture markets (see Section 5.1).

7 Further questions

Open problem 7.1. What are concrete use-cases? See e.g. [12]. For example, what is a class of short, simple programs for which formal verification is economically valuable, and moreover there is value in distributed agreement on a library of software components? The obvious example is smart contracts.

One reason to insist on the framework of first-order logic is that it has been extensively studied, and has a good categorical incarnation in the form of topos theory. Moreover, there is a mature library of tools in the areas of formal verification and automated theorem proving most of which target first-order logic.

Remark 7.2. Hedging is an example of self-insurance and a widely discussed use case for prediction markets. Anyone can use prediction markets to hedge their positions and limit their downside. Software companies insuring themselves against bugs in their software might be a source of capital for conjecture markets.

[redacted]

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]

A Geometric theories

A formula ϕ in a first-order theory language is *geometric* if it can be obtained from atomic formulas by conjunction, disjunction and existential quantification; the name originates from the fact that geometric morphisms $f : \mathcal{E} \longrightarrow \mathcal{F}$ have the property that the left adjoint f^* is compatible with the interpretation of the "set comprehension" $\{x \mid \phi\}$ as an object of $\mathcal{E}, \mathcal{F}$ [1, Theorem X.5]. A theory T is *geometric* if all the axioms are of the form

$$\forall x_1 \ldots \forall x_n (\phi(x_1, \ldots, x_n) \Longrightarrow \psi(x_1, \ldots, x_n))$$

where ϕ, ψ are geometric. In the category theory literature, geometric formulas are sometimes called *positive existential* formulas [22, p.45] and there is a well-understood connection between geometric theories, sketches and accessible categories.[6] For the reader with a computer science background, we mention that geometric logic extends Horn clause logic, on which the programming language Prolog is based [24].

According to Johnstone "It is remarkable how few of the first-order theories encountered in the practice of mathematics fail to be (at least classically equivalent to) coherent theories" [11, §D.1.1]. All algebraic theories, such as group theory and ring theory, are geometric, as are all essentially algebraic theories, such as category theory, the theory of fields, local rings, lattice theory, projective geometry, separably closed local rings. If we allow infinite disjunctions then the infinitary theory of torsion abelian groups is geometric. However it is certainly not true that every first-order theory of mathematical interest is geometric: for example, an infinitary first-order theory which is *not* geometric is given by the theory of metric spaces [11, Example 1.1.7(l)].

Further, it is not clear that every natural software specification can be written down as a geometric theory. It is therefore an important fact that every first-order theory T has a conservative geometric extension T' [26, Theorem 7.7]. This extension is finite and can be implemented on a computer, and the models of T, T' are essentially the same (although the notion of *morphism* is changed, so the categories are not equivalent).[7] The feasibility of a proof assistant based around geometric theories hinges, therefore, on a close analysis of how cumbersome it is to deal with this geometric extension in practice.

The good news is that effective theorem-proving for geometric theories can be automated [23, 24, 25, 27, 31, 30, 28]. For recent practical applications of geometric logic based automatic theorem provers see [32, 33] which use the ArgoCLP system described in [28]. Moreover proofs in geometric logic are "more readable", according to [29]:

> Our proof representation is developed also with readable proofs in mind. Readable proofs (e.g., textbook-like proofs), are very important in mathematical practice. For mathematicians, the main goal is often, not only a trusted, but also a clear and intuitive proof. We believe that coherent logic is very well suited for automated theorem proving with a simple production of readable proofs.

It remains unclear to how appropriate geometric theories are in the context of software specifications, however Vickers has argued in favour of such an approach [5, 6].

[6] Note that many authors refer to such theories as *coherent* and use the term geometric for more general theories which allow infinite disjunctions. Since [1] was the main reference for our seminar, we prefer to stick to this terminology.

[7] A related construction, referred to as the *Morleyization* of T in [11, D1.5.13], produces a geometric theory T' whose category of models in **Sets** (or more generally in any Boolean coherent category) is the same as that of T. This involves adding infinitely many new symbols to the language of T, and is hardly reasonable from the point of view of automating reasoning in a proof assistant.

References

[1] S. MacLane and I. Moerdijk, *Sheaves in geometry and logic: A first introduction to topos theory*, Springer-Verlag 1992.

[2] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2008.

[3] D. Murfet, *The Rising Sea seminar on topos theory and higher-order logic*, http://therisingsea.org/post/seminar-ch/.

[4] J. Lambek and P. J. Scott, *Introduction to higher-order categorical logic*, Vol. 7. Cambridge University Press, 1988.

[5] S. Vickers, *Geometric logic in computer science*, Theory and Formal Methods 1993. Springer, London, 1993. pp.37–54.

[6] S. Vickers, *Geometric logic as a Specification Language*, Theory and Formal Methods, 1994.

[7] B. Potter, D. Till and J. Sinclair, *An introduction to formal specification and Z* , Prentice Hall PTR, 1996.

[8] G. Klein et al. *seL4: Formal verification of an OS kernel*, Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009.

[9] T. Murray and P .C‘.van Oorschot, *BP: Formal Proofs, the Fine Print and Side Effects*, IEEE SecDev 2018.

[10] E. Dennis, A. Kitaev, A. Landahl and J. Preskill, *Topological quantum memory*, Journal of Mathematical Physics 43.9 (2002): 4452–4505.

[11] P. T. Johnstone, *Sketches of an elephant: A topos theory compendium* Vol. 1. Oxford University Press, 2002.

[12] K. Claessen, R. Hähnle, and J.Mortensson, *Verification of hardware systems with first-order logic*, Proceedings of the CADE-18 Workshop-Problem and Problem Sets for ATP. No. 02/10. 2002.

[13] R. Hanson, *Bets As Signals of Article Quality*, October 26, 2018, http://www.overcomingbias.com/2018/10/bets-as-signals-of-article-quality.html

[14] R. Hanson, *How to fund prestige science*, November 10, 2018, http://www.overcomingbias.com/2018/11/how-to-fund-prestige-science.html#more-31927.

[15] R. Hanson, *Idea Futures*, http://mason.gmu.edu/ rhanson/ideafutures.html.

[16] P. Zadarnowski, *C, Lambda Calculus and Compiler Verification - a study in Haskell of purely-functional techniques for a formal specification of imperative programming languages and an epistemically-sound verification of their compilers*, PhD dissertation, University of New South Wales, Sydney, Australia, 2011.

[17] M. Dudik, S. Lahaie, M. D. Pennock, *A tractable combinatorial market maker using constraint generation*, In Proceedings of the 13th ACM Conference on Electronic Commerce (pp. 459-476). ACM, 2012.

[18] Y. Chen and D. M. Pennock, *A utility framework for bounded-loss market makers*, In Proc. of UAI, pages 349358, 2007.

[19] R. Hanson, *Logarithmic markets coring rules for modular combinatorial information aggregation*, The Journal of Prediction Markets, 1(1), 3-15, 2012.

[20] D. M. Pennock and L. Xia, *Price updating in combinatorial prediction markets with Bayesian networks*, arXiv preprint arXiv:1202.3756.

[21] S. Garrabrant, T. Benson-Tilsen, A. Critch, N. Soares and J. Taylor, *Logical induction*, arXiv preprint arXiv:1609.03543, 2016.

[22] M. Makkai and R. Paré, *Accessible categories: the foundations of categorical model theory* (Vol. 104). American Mathematical Soc., 1989.

[23] M. Bezem, T. Coquand, *Automating coherent logic*, Proceedings of LPAR 2005, LNCS **3835**, pp. 246–260, Springer, 2005.

[24] M. Bezem, T. Coquand and A. Waaler, *Research proposal: automating coherent logic*, , 2006.

[25] M. Bezem, T. Hendricks, *On the mechanization of the proof of Hessenbergs theorem in coherent logic*, Journal of Automated Reasoning **40**, pp 61–85, 2008.

[26] R. Dyckhoff and S. Negri, *Geometrisation of first-order logic*, Bulletin of Symbolic Logic 21.2 (2015): 123-163.

[27] J. Fisher and M. Bezem, *Skolem machines*, Fundamenta Informaticae **91**, pp 79-103, 2009.

[28] S. Stojanović, V. Pavlović and P. Janičić, *A coherent logic based geometry theorem prover capable of producing formal and readable proofs*, Proceedings of Automated Deduction in Geometry 2010, LNAI 6877, pp 201220, Springer, 2011.

[29] S. Stojanović, J. Narboux, M. Bezem and P. Janičić, *A vernacular for coherent logic*, In Intelligent Computer Mathematics (pp. 388-403). Springer, Cham., 2014.

[30] H. de Nivelle, J. Meng, *Geometric Resolution: a proof procedure based on finite model search*, Proceedings of IJCAR 2006, LNAI 4130, pp 303-317, Springer, 2006.

[31] B. Holen, D. Hovland and M. Giese, *Efficient rule-matching for hyper-tableaux*, 9th International Workshop on Implementation of Logics Proceedings, Easy-Chair Proceedings in Computing Series 22, Easy-Chair, pp 417, 2013.

[32] V. Marinković, *Proof simplification in the framework of coherent logic*, Computing and Informatics, 34(2), 337-366, 2015.

[33] S.Durdević, J.Narboux and P. Janičić, *Automated generation of machine verifiable and readable proofs: A case study of Tarskis geometry*, Annals of Mathematics and Artificial Intelligence, 74(3-4), 249-269, 2015.

Department of Mathematics, University of Melbourne
E-mail address: d.murfet@unimelb.edu.au