Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики
Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Разработка модели криптовалюты на основе алгоритма консенсуса реализующего метод доказательства доли владения

Агапов Г.Д.
Научный руководитель: Чивилихин Д.С.

Санкт-Петербург
2018

ОГЛАВЛЕНИЕ

Стр.

ВВЕДЕНИЕ

TODO Введение

ГЛАВА 1. **ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ**

В настоящей главе проводится краткий обзор предметной области.

Вводится понятие блокчейна, рассматриваются консенсус-алгоритмы, используемые в блокчейн-системах. Вводится понятие криптовалюты, рассматриваются реализованные системы, реализующие функционал криптовалюты, спецификации к ним.

1.1. Блокчейн

Опр. 1.1. Блокчейн (в переводе с английского ”цепочка блоков”) – структура данных, представляющая собой последовательный непрерывно расширяющийся список записей, связанных между собой по принципу односвязного списка, с использованием криптографически стойкой хэш-функции для установления связей.

Запись в блокчейне вместе с её связью принято называть блоком. В реализациях структуры блокчейн, как правило, каждый связью является ссылка на предыдущий блок, где ссылкой является значение криптографически стойкой хэш-функции, примененной к предыдущему блоку.

$$
\begin{aligned}
&Blockchain_{\langle H, Data\rangle} = \{\ \langle origin, blocks\rangle \mid origin \in Data,\\
&\ blocks \in List_{Data\times Value_H},\\
&\ \langle index, block \equiv \langle data, value_H\rangle\rangle \in blocks\\
&\qquad \Longrightarrow \begin{cases} value_H = H(origin), & \text{if } index = 1\\ value_H = H(blocks[index-1]), & \text{otherwise}\end{cases}\\
&\}
\end{aligned}
\tag{1.1}
$$

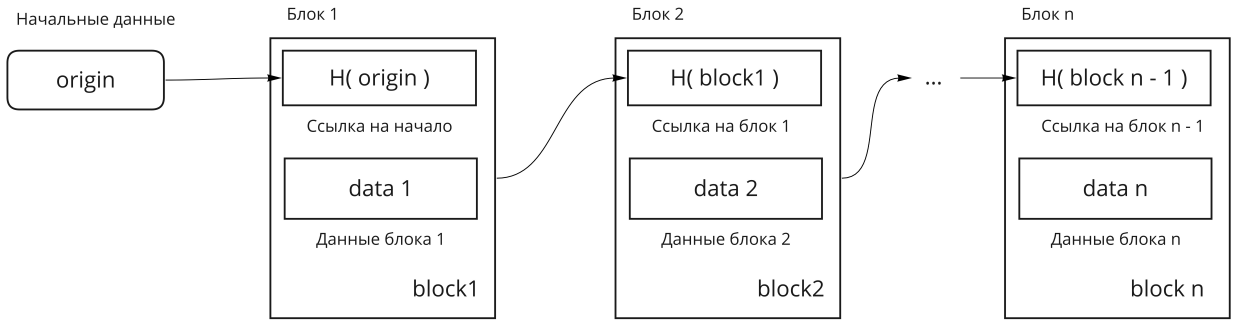

Рис. 1.1 — Диаграма структуры Блокчейн

Построенная таким образом структура данных имеет интересное свойство по сравнению с односвязным списком:

$$
\begin{aligned}
&\forall i, n, x_0, ..., x_n, x'_i \\
&\langle x_0, [\ block_1 \equiv \langle x_1, H(x_0)\rangle, ..., \\
&\qquad block_i \equiv \langle x_i, H(block_{i-1})\rangle, ..., \\
&\qquad block_n \equiv \langle x_n, H(block_{n-1})\rangle\ ]\rangle \in Blockchain_{\langle H, Data\rangle}, \\
&i \neq n, x_i \neq x'_i \implies
\end{aligned}
\tag{1.2}
$$

$$
\begin{aligned}
&\langle x_0, [\ block_1 \equiv \langle x_1, H(x_0)\rangle, ..., \\
&\qquad block'_i \equiv \langle x'_i, H(block_{i-1})\rangle, ..., \\
&\qquad block_{i+1} \equiv \langle x_{i+1}, H(block_i)\rangle, ..., \\
&\qquad block_n \equiv \langle x_n, H(block_{n-1})\rangle\ ]\rangle \notin Blockchain_{\langle H, Data\rangle},
\end{aligned}
$$

Иными словами, невозможно поменять содержание блока, не меняя при этом последующих блоков в структуре. Данное свойство опирается на предположении о стойкости криптографической функции H, в частности: для данного $h \in Value_h$ практически невозможно подобрать x такой что $H(x) = h$.

1.1.1. Применение структуры Блокчейн

Гораздо чаще термин “Блокчейн” применяется не к структуре как к таковой, а к классу систем, использующих её как одну из основных компонент. Большая часть известных систем, использующих эту структуру, представляют из себя распределенные базы данных, как правило специализированные под конкретные области применения. Примеры таких систем:

– Криптовалюты (распределенная база данных, предназначенная для хранения информации о счетах, проведения транзакций над ними).
  – Примеры: Bitcoin [1], NXT [2], Cardano [3].
– Системы для распределенных вычислений с использованием т.н. смарт-контрактов.
  – Как частный случай распределенных вычислений, используются для реализации финансовых контрактов.
  – Примеры: Ethereum [4], NEO [5].
– Распределенные базы данных.
  – Как правило, реализуются для хранения данных, относящихся к конкретной области применения
  – Примеры: Namecoin [6] (альтернатива системе DNS), Disciplina [7] (система для хранения и обмена данных об академической успеваемости)

Структура Блокчейн нашла такое широкое применение в построении распределенных баз данных в первую очередь в следствии свойства 2.1. Если участники сети достигли консенсуса (возможно, в нестрогом смысле) относительно блока b, то они автоматически находятся в солгасии со всеми блоками, предшествующим b в структуре.

1.2. Консенсус-алгоритмы в блокчейн-системах

Большинство алгоритмов консенсуса, разработанных для блокчейн-систем являются алгоритмами для достижения в системе согласованности в конечном счёте (англ. eventual consistency) или эвентуального консенсуса. В отличие от классических консенсус алгоритмов, таких как Paxos [8], Raft [9], от алгоритма консенсуса не требуется достижение полного консенсуса в системе, но полагается достаточным достижение частичного консенсуса, переходящего в полный с вероятностью увеличивающейся со временем с момента достижения согласия о некотором значении большинством узлов сети.

Первопроходцем среди распределенных систем, использующих структуру Блокчейн по праву считается криптовалюта Bitcoin [1]. В статье “Bitcoin: A Peer-to-Peer Electronic Cash System” за авторством Satoshi Nakamoto [10],

описывающей концепцию децентрализованной системы электронных денег, предлагается использование метода доказательства выполнения работы (англ. proof of work) для построения алгоритма эвентуального консенсуса. В оригинальной статье предлагается алгоритм евентуального консенсуса без подробного анализа, в последующих работах других авторов, в частности “The bitcoin backbone protocol: Analysis and applications” [11], доказано что предложенный алгоритм (с небольшими модификациями) является толерантным к атакам, если как минимум 2/3 вычислительной мощности сети находится в управлении честных участников.

Метод доказательства выполнения работы был в дальнейшем применен для построения алгоритмов эвентуального консенсуса для систем Ethereum [4], ZCash [12], IOTA [13], равно как и множества других блокчейн-систем.

Другим популярным методом для построения консенсус-алгоритмов в блокчейн-системах является метод доказательства доли владения (англ. proof of stake), нашедший своё применение в построении алгоритмов эвентуального консенсуса Ouroboros [14], Snow White [15], алгоритмов для систем NXT [2], NEO [5]. Метод может быть также применен в построении алгоритмов полного консенсуса, в частности в алгоритме Algorand [16], также возможность применения метода доказательства доли владения для разработки распределенных систем упомянута авторами алгоритма Honey Badger BFT [17].

1.2.1. Метод доказательства выполнения работы

TODO описания принципа PoW

1.2.2. Метод доказательства доли владения

TODO описания принципа PoS

1.3. Криптовалюты

Опр. 1.2. Криптовалюта – собирательный термин для систем электронных денег, использующих методы криптографии для установления собственности средств при проведении транзакций.

В определении, данном, выше, заложен следующий аспект большинства (если не всех) систем, относимых к криптовалютам: использование асинхронной криптографии. Каждая денежная единица в системе ассоциирована с некоторым адресом `Addr = CreateAddr ( PubKey )`, построенном на основе публичного ключа `PubKey`. Для проведения трансфера средств с одного адреса на другой (транзакции) требуется предоставить подпись информации о транзакции, сделанной секретным ключом `SecretKey`, соответствующим публичному ключу `PubKey`.

1.3.1. Функциональность криптовалюты

Базовой функциональностью криптовалюты, является отправление транзакций между различными участниками сети: участник S имеет возможность послать средства на сумму c участнику T при условии что ему известен адрес участника T и на его адресах содержится средств на сумму, не меньшую c. Однако, уже в первой криптовалюте, нашедшей широкое распространение, Bitcoin [1], создателями было реализовано множество дополнительных функциональностей, в частности:

- Поддержка комиссий за проведение транзакций
- Поддержка скриптовых транзакций
- Поддержка транзакций типа M из N
- Поддержка децентрализованной пириноговой сети

Комиссии за проведения транзакций является важным механизмом для противодействия атакам отказа в доступе, равно как и механизмом поощрения участников, поддерживающих сеть (выпускающих блоки).

Поддержка скриптовых транзакций представляет собой возможность создания адреса, правила валидации права владения которым задаются при помощи предоставляемой пользователем программы (скрипта), запускаемой участниками сети, поддерживающими сеть, при проверки транзакции, использующей средства с данного адреса. Скриптовые транзакции являются очень мощным механизмом, позволяющие реализовать семантику многих составных финансовых операций. Однако, возможности использования скриптовых транзакций для построения финансовых контрактов с помощью скриптового языка в Bitcoin [1] весьма ограничены, т.к. семантика использования скриптовых транзакций не ползволяет описать конечный автомат (только ко-

нечный автомат, выполняющий ровно один переход из начального состояния в конечное). Это ограничение было устранено в системе Ethereum [4].

Поддержка транзакций типа M из N является частным случаем финансового контракта. Задача состоит в предоставлении N участникам сети возможности создания адреса A для списания средств с которого требуется взаимодействие как минимум M участников из N, участвоваших в создании адреса. В простейшем случае, адрес A будет представлять из себя конкатенацию публичных ключей N участников и предоставление подписей транзакции M участниками требуется для валидации транзакции. Именно так транзакции типа M из N реализованы в Bitcoin [1], при использовании скриптового языка. Однако возможны и приницпиально отличные реализации этого типа транзакций, например с использованием пороговых криптосистем [18].

Большинство криптовалют разрабатываются как открытые децентрализованные пиринговые (Peer-to-peer или P2P) сети, т.е. позволяют любому пользователю сети Интернет стать участником сети. Однако это не является необходимым требованием для создания криптовалюты. Существуют проекты криптовалют с закрытой топологией сети (например, проект Hyperledger [19]), равно как и централизованные (RSCoin [20]). Создателями Bitcoin протоколы для поддержки пиринговой сети были реализованы с нуля. Создатели сети Ethereum взяли за основу своей сети протокол Kademlia [21], внеся в него ряд модификаций.

Помимо упомянтых выше функциональностей существуют и другие, поддерживаемые во многих блокчейн-системах:

- Делегация доли владения
  - Реализовано во многих криптовалютах, реализующих метод доказательства доли владения, в частности в NEM [22].
- Поддержка обновления протоколов
  - Поддержка “ручного” обновления протоколов (достижения согласия между участниками сети) реализована в большинстве блокчейн-систем посредством, как правило, версионирования форматов блока, протоколов сети и установления процедур коммуникации между командами людей, поддерживающих узлы сети.

    - Поддержка полностью автоматизированных обновлений протокола анонсирована для криптовалюты Tezos [23], не поддерживается подавляющим большинством блокчейн-систем.
- Поддержка обновления программного обеспечения участников сети.
- Расширения понятия блокчейна до понятия ациклического графа блоков
    - Реализовано в криптовалюте IOTA [13].
- Поддержка криптографических протоколов доказательства с нулевым разглашением [24]
    - Реализовано в криптовалюте ZCash [12]

1.3.2. Спецификации существующих систем

Стандартом блокчейн-индустрии является написание технической спецификации, часто форматируемой по типу научной публикации, задача которой состоит в описании основных технических решений системы.

В таблице 1.1 рассмотрены технические спецификации некоторых известных криптовалют с точки зрения детализации, содержания.

Следует заметить что большинство существующих блокчейн-систем не предоставляют должного качества спецификаций, что сужает возможность их анализа с целью рассмотрении их модели. Многие системы являются клонами или несущественными (с точки зрения проблем, рассматриваемых в настоящей работе) модификациями систем, рассмотренных в таблице 1.1. Документы, упомянутые в таблице 1.1 являются одними из лучших, имеющихся в доступе.

Таблица 1.1 — Сравнение документации некоторых криптовалют

Документ	Детализация	Комментарий
Ethereum yellow paper [25]	Высокая	Спецификация консенсус протокола вынесена в отдельную статью. В подробной форме специфицирована логика валидации транзакций, блоков.

Документ	Детализация	Комментарий
Bitcoin developer guide [26]	Средняя	Многократно подчеркнуто, что документация не является спецификацией и для спецификации следует обращаться к реализации клиента `bitcoind`.
EOS.IO Technical White Paper [27]	Низкая	Документ покрывает только основные идеи механизмов, используемые в системе EOS. Для описанных механизмов не даётся подробной спецификации, только высокоуровневый обзор.
Peercoin [28]	Низкая	В документе описывается в неформальной форме консенсус-алгоритм, не приводится никакой математической модели алгоритма, равно как и анализа. В документе не описывается иные компоненты криптовалюты (помимо консенсус-алгоритма).
NEM technical reference [22]	Средняя	Документ является подробной технической спецификацией компонент системы NEM. Спецификация не предлагает общей модели, описывая компоненты раздельно. Кроме того, логика валидации опускается (т.е. предполагается что её можно извлечь из описаной семантики, что является затруднительным).

1.4. Задача построения модели криптовалюты

Как было показано в главе 1, на сегодняшний день разработано большое количество блокчейн-систем. Каждая из этих систем реализует различные наборы функциональностей криптовалюты, причем реализации могут сильно отличаться, равно как и способы, с помощью которых функциональности взаимодействуют друг с другом. Создатели очередной блокчейн-системы зачастую вынуждены решать задачу построения системы “с нуля”, опыт коллег может быть использован лишь на уровне заимствования идей, возможности анализа последствий использования этих идей в построенной системе значительно ограничены. В равной степени затруднен и анализ новых идей, желание реализовать которые часто и является отправной точкой для разработки новой системы.

Многие подходы, используемые в блокчейн-системах, опираются на достижения криптографии, использование которых накладывает на разработчика системы ответственность за четкое понимание требований, предъявляемых используемой структурой. То же относятся и к более сложным конструкциям, использующим криптографические примитивы как составную часть. Недостаточное понимание требований, предъявляемых используемыми конструкциями при разработке новых систем может легко привести к проблемам с безопасностью, производительностью.

Анализ разработанных систем, равно как и работа над дизайном новых значительно осложняется отсутствием формализации функциональностей криптовалюты, описанием способов из объединения в общую систему. Эти и другие затруднения в разработке дизайна новых блокчейн-систем, равно как и анализе существующих, можно решить посредством формализации понятия криптовалюты, разработки единой абстрактной модели. Функциональности криптовалюты могут быть описаны как составные компоненты разработанной модели.

Отдельно стоит вопрос разработки блокчейн-систем, использующих метод доказательства доли владения. До публикации работ Ouroboros [14], Snow White [15], Algorand [16] в 2016, не было ни одного доказанно безопасного алгоритма консенсуса (эвентуального или полного), использующего метод доказательства доли владения. Для некоторых существовавших на тот

момент алгоритмов были применены атаки [29], против которых по утверждению авторов алгоритмов, алгоритмы должны были быть устойчивыми. В отличие от предшествоваших работ, в Oroboros (который является первым опубликованным доказанно безопасным алгоритмом, использующим метод доказательства доли владения) авторами строится модель среды, в которой формулируется алгоритм и доказывается его устойчивости при работе в данной среде к известным на момент создания работы атакам.

Однако в вопросе построения блокчейн-систем, на сегодняшний день нет работ, предлагающих формальную модель системы, основанной на консенсус-алгоритме, реализующем метод доказательства доли владения, формулизующих свойства модели и доказывающих их. Требуется заметить, что консенсус-алгоритм является в некотором смысле лишь одной из функциональностей криптовалюты и как следствие этого, даже построив систему на основе доказанно безопасного алгоритма консенсуса, ничего нельзя сказать о безопсаности системы в целом.

1.4.1. Критерии для построенной модели

Для того чтобы перейти от обоснования востребованности задачи построения модели криптовалюты, реализующей метод доказательства доли владения, к её непосредственной формулировке, рассмотрим критерии, которым должна удовлетворять построенная модель, позволяющие эффективно использовать модель в анализе существующих и разработке новых блокчейн-систем (криптовалют в частности):

- Модульность
  - Каждая функциональность должна быть представлена как набор компонент модели, описанных независимо и предъявляющие друг к другу некоторый набор требований.
- Детализированность
  - Всякое сформулированное войство должно быть либо доказано для приведенной конструкции, либо вынесено в требования для компоненты, используемой этой конструкцией.
- Эффективность
  - Для разрабатываемой модели должна быть показана эффективность её реализации на практике, причем эффективная реализа-

ция должна лишь реализовывать требования модели, не изменяя составляющие.

- Безопасность
  - Система, разработанная на основе модели, должна отвечать требованиям безопасности, в частности быть устойчивой к известным атакам.
  - Система, разработанная на основе модели, должна иметь резистентность к взлому одного из используемых в ней криптографических примитивов.

Требование безопасности является особенно важным в разработке надёжных финансовых систем (что является одним из освновных применений криптовалют на сегодняшний день).

Требования детализированности и модульности должны облегчить задачу проведения анализа модели, равно и расширения модели новыми компонентами.

Требование эффективности важно при конструировании модели, игнорирование требования на ранних этапах может привести к получению модели, неприменимой на практике (следовательно, представляющий малый интерес в вопросе проектирования новых и анализа существующих систем).

1.4.2. Существующие попытки построения модели криптовалюты

Во всех документах, расмотренных в таблице 1.1 приводится техническое описание (разной степени детализации) системы в целом, но ни в одной из них не была предпринята попытка модульного (поэтапного) построения формальной модели. Приведенные же модели, даже в случае достижения высокой степени детализации, слабо подходят для дальнейшего анализа в силу своей монолитности, большого объёма.

Авторы статьи, предлагающей дизайн криптовалюты Tezos ([23, 2.1 Mathematical representation]) предлагают некоторую математическую нотацию для формирования модульной модели криптовалюты, однако используют её только для описания понятия глобального состояния и блока как изменения этого состояния (самых базовых понятий), в дальнейшем описании дизайна системы нотация не используется. Авторы технической специфика-

ции Ethereum [25] достаточно подробно вводят формулировки состояния, однако все структуры, равно как и логика валидации, описаны как части одной цельной (и довольно громоздкой) модели, без модульной структуры.

В статьях, посвященных описанию и анализу консенсус алгоритмов (не криптовалют целиком), таких как Ouroboros [14], Algorand [16], авторами предлагается только математическая модель консенсус-алгоритма, к остальным компонентам системы предъявляется набор требований (но описание этих компонент выходит за рамки тем данных работ).

Фреймворк Scorex

Авторы фреймворка Scorex [30] поставили своей целью создание модульного фреймворка для прототипирования различных блокчейн-систем. Фреймворк разработан на языке Scala и сопровожден достаточно подробной документацией [31], объясняющей основные концепции, стоящие за фреймворком. Авторы фреймворка отдельно выделяют транзакционный уровень, уровень консенсуса. Это единственная известная автору попытка построения модели криптовалюты в общем смысле.

Как негативный фактор, в документации описываются только некоторые отдельные компоненты модели. Сведение отдельных компонент в единую модель в документации отсутствует. Код фреймворка на Scala, впрочем может рассматриваться как модель криптовалюты (в достаточно общем смысле). Однако, следует заметить, реализация многих компонент Scorex не является абстрактным обобщением конструкций, но является упрощенной реализацией таковых, позволяющей использовать их в прототипировании других компонент.

Модели криптовалют с PoS

Автору неизвестны детальные модели криптовалют, построенные для консенсус-алгоритмов, реализующих метод доказательства доли владения (PoS).

В фреймворке Scorex [30] приведена лишь абстрактная модель, без специализации относительно метода PoS. Кроме того, отсутствие последовательной формулировки модели в документации является минусом (код на Scala

часто слишком специализирован, чтобы использовать его для построения абстрактной модели, не следующей определениям аналогичным принятым в коде).

Спецификация Ethereum достаточно детальна, но не предлагает модульной модели и кроме того относится к системе, использующей консенсус-алгоритм, реализующий метод доказательства выполнения работы.

Спецификации систем, реализующих метод доказательства доли владения EOS [27], NEM [22], Peercoin [28] не удовлятворяют требованиям детализации. Спецификация криптовалюты NEM выделяется сравнительно полной детализацией, однако авторами не приводится формальная модель. Кроме того, в последующих главах будет указано на несколько несогласованностей в дизайне этой криптовалюты.

1.5. Цель и задачи настоящей работы

Рассмотрение модели криптовалюты следует начать с ответа на вопрос, что таковая модель должна в себя включать. Всякую блокчейн-систему можно разделить на несколько уровней:

- Уровень взаимодействия между узлами сети
  - Включает в себя в частности сетевые протоколы, форматы сериализации.
  - Отвечает за обмен данными между узлами сети.
- Логика обработки данных
  - Включает в себя все правила изменения состояния системы, в частности валидацию предложенных другими узлами сети (пользователем) данных.
- Пользовательская функциональность
  - Включает в себя функционал, требующийся пользователю системы для эффективного с ней взаимодействия.

В настоящей работе рассматривается в первую очередь логика обработки данных, т.к. она является базовой при рассмотрении систем баз данных, частным случаем которых являются блокчейн-системы.

Цель настоящей работы: разработать модель обработки данных криптовалюты, использующую алгоритм консенсуса, реализующий метод доказательства доли владения.

Задачи настоящей работы:

1. разработать модель блокчейн-системы общего вида;
2. разработать модель блокчейн-системы, использующей метод доказательства доли владения;
3. разработать модель криптовалюты;
4. показать соответствие разработанных моделей критериям, сформулированным в разделе 1.4.1.

Разработанная модель блокчейн-системы, использующей метод доказательства доли владения должна включать в себя:

– механизм делегации доли владения;
– механизм обновления протокола.

Разработанная модель обработки данных должна позволять эффективную реализацию уровней взаимодействия между узлами сети, пользовательского взаимодействия.

Резюме

В данной главе был проведен обзор предметной области блокчейн-систем, криптовалют в частности. Рассмотрены основные подходы к построению консенсус-алгоритмов в них. Приведены результаты анализа спецификаций существующих блокчейн-систем, существующих попыток построения модели блокчейн-систем.

Сформулированы цель и задачи настоящей работы.

ГЛАВА 2. МОДЕЛЬ БЛОКЧЕЙН-СИСТЕМЫ

2.1. Модель состояния системы

Как было замечено в разделе 1.1.1, большинство блокчейн-систем представляют из себя распределенные базы данных. В следствии этого, для построения модели блокчейн-системы, следует начать с формализации состояния базы данных, способа его изменять.

```
1 import qualified Data.Map as M
2 import Data.Map (Map)
3
4 newtype Prefix = Prefix Int
5 type Prefixed k = (Prefix, k)
6
7 type StateP id value = Map (Prefixed id) value -- Portion of state
```

Листинг 2.1 — Структура состояния системы

Тип `StateP` выражает тип состояния системы. Причем как глобального состояния целиком, так и его части. Состояние представляется как словарь, отображение из ключа `(Int, id)` в значение `value`. Целочисленный префикс ключа введен для удобства дальнейшего построения.

И ключ, и префикс введены как абстрактные типы. Это означает, что для описания модели нам не требуется указывать, какой именно тип имеет ключ или значение. При описании работы с состоянием (в дальнейшем), будут накладываться различные ограничения (требования) на ключ или значение. Для выражения ограничений в нотации будет использоваться конструкция языка Haskell тайп-класс, например:

```
1 getMonoidValue :: (Ord id, Monoid value)
2              => Prefixed id -> StateP id value -> value
3 getMonoidValue key stateP = maybe mempty (M.lookup key stateP)
```

Листинг 2.2 — Пример наложения ограничений на абстрактные переменные

В примере было наложено ограничение `Ord id` на ключ, выражающее требование существования линейного порядка на множестве всех ключей. В сигнатуре функции указано еще одно требование, `Monoid value`, требующее от `value` моноидальную структуру.

2.1.1. Вид транзакции

В листинге 2.3 формализовано изменение состояния. Изменение словаря можно представить как тип-произведение из множества ключей, которые требуется удалить `csRemove` и множества пар ключ-значение, которые требуется добавить в словарь `csAdd`.

```
data ChangeSet id value = ChangeSet
   { csAdd :: Map (Prefixed id) value
   , csRemove :: Set (Prefixed id)
   }
```

Листинг 2.3 — Изменение состояния системы

Применение объекта типа `ChangeSet id value` к словарю должно иметь следующую семантику:

$$applyChangeSet\ (\ state : StateP,\ \langle csAdd, csRemove\rangle : ChangeSet\ )\ \ = \begin{cases} \bot, & \text{if} \\ & csRemove \cap keys(state) \neq csRemove \\ & \quad \vee (keys(csAdd) \setminus csRemove) \cap keys(state) \neq \emptyset \\ (state \setminus csRemove) \cap csAdd, & \text{otherwise} \end{cases} \tag{2.1}$$

Что замечательно, тип `Either e (ChangeSet id value)` является моноидом, где `e` – некоторый тип ошибки. Это позволяет объединять множество транзакций в одно изменение типа `ChangeSet id value`, применяемое в дальнейшем к состоянию систему. Это свойство полезно для построения эффективной системы.

Транзакция к состоянию имеет следующий вид:

```
newtype StateTxType = StateTxType Int

data StateTx id proof value = StateTx
   { txType :: StateTxType
   , txProof :: proof
   , txBody :: ChangeSet id value
   }
```

Листинг 2.4 — Транзакция

Помимо изменения состояния, транзакция содердит в себе еще два поля: `txType` и `txProof`. Поле `txType` представляет из себя целочисленный идентификатор типа транзакции. Поле `txProof` содержит в себе дополнительную информацию, требующуюся валидатору для проверки корректности транзакции. В частности, для транзакции, изменяющей баланс пользовательского счёта, `txProof`, должен содержать подпись транзакции секретным ключом пользователя-владельца средств.

Оба поля будут использованы в последствии в построении валидатора. Что существенно, поле `txBody` содержит в себе полный набор изменений, которые будут применены к состоянию системы. Ни `txProof`, ни `txType` не будут применены к состоянию, будут забыты как только валидатор их обработает.

2.1.2. Валидация транзакции

Простейший валидатор транзакции может быть описан функцией, имеющей одну из сигнатур:

```
validator1 :: StateTx id proof value -> Bool
validator1 = ...

validator2 :: StateP id value -> StateTx id proof value -> Bool
validator2 = ...

validator3 :: StateTx id proof value -> (Set id, StateP id value -> Bool)
validator3 = ...
```

Функция `validator1` принимает транзакцию и возвращает `True` тогда и только тогда когда транзакция является корректной. Однако, только для очень небольшого класса транзакций можно выразить валидатор, используя функцию с такой сигнатурой. Реализация практически любой функциональности криптовалюты требует доступа валидатора к состоянию.

Функция `validator2` принимает на вход глобальное состояние системы. Функция позволяет выразить класс транзакций, чьи валидаторы требуют только доступа к состоянию системы для проверки транзакции. Такой класс достаточно широк для построения модели криптовалюты, однако использование такой сигнатуры функции является препятствием для построения эффективной реализации.

В существующих на сегодняшний день системах состояние системы измеряется в гигабайтах данных, передавать состояние как единый объект, полностью загруженный в оперативную память компьютера, является неэффек-

тивным расходом ресурсов. Функция `validator3` в отличие от `validator2` не передаёт глобальное состояние как единый объект, но позволяет запросить интересующие ключи из состояния и возвратить результат валидации транзакции в соответствии с возвращенными значениями.

Однако `validator3`, как и `validator1`, описывает только узкий класс транзакций, в общем смысле для валидации транзакции может потребоваться сделать более одного запроса к глобальному состоянию. Например, для проверки отсутствия циклов в цепочке делегаций, требуется выполнить как минимум `maxDlgHeight` запросов к базе, где `maxDlgHeight` – максимальная путь в ациклическом графе делегации (параметр делегации). Процесс валидации делегационной транзакции будет подробно рассмотрен в разделе **??**.

Введем несколько вспомогательных типов, чтобы выразить тип валидатора, позволяющего выразить класс транзакций, имеющих возможность запроса ключей из состояния произвольное количество раз, допускающего эффективную реализацию. Воспользуемся концепцией свободной монады, определенной в языке Haskell (подробнее с концепцией свободных монад можно ознакомиться в документации к библиотеке free [32]). Свободная монада определяется следующим типом данных:

```
data Free f a
  = Pure a
  | Free (f (Free f a))
```

Листинг 2.5 — Свободная монада

Тип данных `Free` описывает чистую функцию, которая возвращает либо значение, либо запрос к внешнему контексту. Тип `Free` определяет два конструктора: `Pure` и `Free`. Конструктор `Pure` используется для возврата значения, конструктор `Free` для обращения к внешнему контексту с последующим продолжением вычисления. Переменная типа `f` задаёт конкретный формат обращения к внешнему контексту и обработки результата обращения.

```
newtype DataAccess1 req resp res = DataAccess1 (req, resp -> res)
  deriving (Monoid, Functor)
```

Тип `DataAccess1` представляет из себя пару, первым элементом которой является запрос к состоянию, вторым – функция, которая принимает результат исполнения запроса и возвращает функцию продолжения вычисления. Подставив `req = Set id`, `resp = Map (Prefixed id)value`, мы получаем конктрук-

цию, достаточно экспрессивную для реализации валидатора, итеративно рассматривающего различные ключи состояния. В некоторых случаях этого ожет оказаться недостаточным, например если нам требуется запросить не какой-то конкретный ключ, а все ключи, отвечающие какому-либо критерию. Воспользуемся префиксом: добавим возможность сделать запрос, который либо запрашивает множество ключей, либо проитерироваться всем ключам с префиксом p в состоянии.

```
data FoldF a res = forall b. FoldF (b, a -> b -> b, b -> res)

data DbAccess id value res
  = DbQuery (Set (Prefixed id)) (StateP id value -> res)
  | DbIterator Prefix (FoldF (id, value) res)
  deriving (Functor)

type StatePComputation id value = Free (DbAccess id value)

validator4 :: StateTx id proof value -> StatePComputation id value Bool
validator4 = ..
```

Листинг 2.6 — Функтор обращения к состоянию

Тип `DataAccess` представляет из себя тип-сумму, первый конструктор которого эквивалентен `DataAccess1`, второй представляет из себя интерфейс для итерации по множеству ключей. Тип `FoldF` представляет произвольную функцию свёртки: начальное значение `b`, свёртка по очередному элементу `a -> b -> b` и конверсия результата в тип результата `res`.

Тип `StatePComputation`, сигнатура функции `validator4` показывают, как подстановкой `DataAccess` как параметра в тип `Free` можно получить конструкцию, подходящую для описания валидатора. Сигнатура `validator4` позволяет реализации запрашивать ключи из глобального состояния, получать соответствующие им значения, причем делать это столько раз, сколько потребуется.

Окончательный тип валидатора транзакции предложен в листинге 2.7:

```
newtype PreValidator e id proof value =
  PreValidator
    (StateTx id proof value -> ExceptT e (StatePComputation id value) ())

newtype Validator e id proof value =
  Validator (Map StateTxType (PreValidator e id proof value))
```

Листинг 2.7 — Тип валидатора транзакции

Тип `ExceptT e (StatePComputation id value)()` является непосредственным обобщением типа `StatePComputation id value Bool`, позволяющим в случае ес-

ли валидация транзакции закончилась отрицательным результатом, указать ошибку типа `e` (а не просто `False`). Тип `PreValidator` является достаточным для описания логики валидации, в частности при проверке транзакции, он может рассматривать тип транзакции (поле `txType`), возвращать положительный результат в случае если транзакция является корректной и ошибку в случае если предложенный тип транзакции не предназначен для проверки данным валидатором или если транзакция является некорректной.

Однако, для проверки множества транзакций требуется различать, какой валидатор относится к первому типу транзакции, какой ко второму и т.д., т.к. ошибка, возвращенная валидатором при проверки транзакции, тип которой не соответствует типу, рассматриваемым валидатором, не обязательно указывает на некорректность транзакции. Потому в дополнению к типу `PreValidator` введён тип `Validator`, соддержащий в себе отображение из типа транзакции в соответствующий типу валидатор.

На логику проверки транзакции с помощью объекта типа `Validator` накладывается ограничение, что проверяемый тип транзакции содержится в хранящемся в объекте отображении, иначе возвращается ошибка. Это требуется для того, чтобы исключить возможность положительного результата валидации транзакции, чей тип неизвестен валидатору.

В последующем изложении под термином валидатор будет пониматься объект типа `Validator`, описанный типом выше, под термином предвалидатор – объект типа `PreValidator`.

2.1.3. Моноидальная структура валидатора

Важным свойством определенных выше валидатора, предвалидатора является их моноидальная структура. Рассмотрим подробнее определения моноидальной единицы, моноидального умножения для этих двух типов.

Моноидальная единица для предвалидатора реализуется просто: это предвалидатор, разрешающий всякую транзакцию. Аналогично для валидатора: единицей является тип, запрещающий всякую транзакцию (пустой ассоциативный массив из типа транзакции в предвалидатор).

Умножение для валидатора представляет из себя объединение двух ассоциативных массивов из типа транзакции в предвалидатор, причем при сов-

падении ключей, требуется воспользоваться моноидальным умножением для предвалидатора.

Умножение для предвалидатора сводится к умножению двух вычислений типа `ExceptT e (StatePComputation id value)()`. Или, эквивалентно, `StatePComputation id value (Either e ())`. Соответствующими типу значениями могут быть: `Free dataAccess`, `Pure (Left e)`, `Pure (Right ())`. Если одно из значений `Pure (Left e)`, вычисление завершается ошибкой `e`, т.е. результат умножения – `Pure (Left e)`. Если один из операндов умножения `Pure (Right ())`, результат умножения равен другому операнду.

Наконец, если мы имеем операнды `Free dataAccess1`, `Free dataAccess2`, следует произвести следующее:

- Если `dataAccess1 = DBQuery idSet1 cont1`, `dataAccess2 = DBQuery idSet2 cont2`, результат умножения будет равен `DBQuery idSet3 cont3`, где $idSet3 = idSet1 \cup idSet2$, $cont3 = \lambda values.cont1\ (values \cap idSet1) \cdot cont2\ (values \cap idSet2)$ (обозначение $\cdot$ используется для моноидального умножения).
- Если `dataAccess1 = DBIterator p (FoldF (init, foldf, resF))`, результат умножения будет `DBIterator p (FoldF (init, foldf, resF'))`, где $resF' = \lambda b.\ (resF\ b) \cdot (Free\ dataAccess2)$
- Аналогично для `dataAccess2 = DBIterator p f2`

Определенное таким образом моноидальное умножение обладает интересным свойством: соседствующие операнды `DBQuery idSet cont` будут объединены в один. Это означает что множество небольших запросов к состоянию будут объединены в один композитный запрос (что представляет собой распространенную и достаточно эффективную оптимизацию при работе с базами данных).

В дальнейшем при построении модели мы будем активно пользоваться моноиальной структурой валидатора и предвалидатора. Эта особенность позволяет описывать различные функциональности независимо друг от друга (посредством описания соответствующих предвалидаторов, валидаторов), а затем совмещать их вместе, используя моноидальное умножение.

2.1.4. Валидация структуры транзакции

Для проверки структуры транзакции на корректность (безотносительно к типу транзакции) следует выолнять проверки свойств, описанных в формуле 2.1. Выполнение этих проверок объединим в предвалидатор `structuralPrevalidator`, которым мы воспользуемся в дальнейшем построении модели.

2.2. Модель блокчейна

Данный раздел посвящен понятию блока. В отличие от большинства попыток формализовать модель блокчейн-системы, настоящая модель оперирует в первую очередь с состоянием системы и его изменением. Блок вводится как сущность, позволяющая описать:

- Цепь последовательных изменений
- Функцию выбора между двумя произвольными цепями изменений
- Ассоциацию множества последовательных изменений, которые требуется применить атомарно
  - Требуется для реализации многих алгоритмов консенсуса, в частности алгоритмов на основе методов доказательства доли владения, доказательства выполнения работы.

2.2.1. Валидация цепи блоков

Рассмотрим тип блока:

```
data Block header payload = Block
   { blkHeader :: header
   , blkPayload :: payload
   }
```

Листинг 2.8 — Структура блока

Блок включает в себя заголовок с абстрактным типом `header` и набор изменений состояния с абстрактным типом `payload` (в дальнейшем “тело блока”).

Что важно, для построения функциональности процессинга блоков нам не требуется информация о конкретном виде состояния, транзакции (описанные

```haskell
newtype BlockIntegrityVerifier header payload e =
    BIV { unBIV :: Block header payload -> Either e () }

data BlkConfiguration header payload blockRef e = BlkConfiguration
   { bcBlockRef :: header -> blockRef
   , bcPrevBlockRef :: header -> Maybe blockRef
   , bcBlkIntegrityVerify :: BlockIntegrityVerifier header payload e
   , bcIsBetterThan :: OldestFirst [] header -> OldestFirst [] header -> Bool
   , bcMaxForkDepth :: Int
   }
```

Листинг 2.9 — Конфигурация валидации цепи блоков

в разделе 2.1). Логика валидации цепи блоков, конфигурация которой представлена в листинге 2.9, не требует доступа к состоянию системы.

Задачи логики валидации цепи:

- Проверить, что предложенная цепочка является корретной
  - Используются методы `bcBlockRef`, `bcPrevBlockRef`
- Целостность блока с точки зрения логики блокчейн-системы
  - Используется метод `bcBlkIntegrityVerify`
- Принять решение о замене принятой на текущий момент цепи $c1$ на предложенную цепь $c2$
  - Используются методы `bcIsBetterThan`, `bcMaxForkDepth`

Функция `bcBlockRef` позволяет получить уникальный идентификатор блока `blockRef`, как правило для его получения используется криптографически стойкая хэш-функция, применяемая к блоку. Функция `bcPrevBlockRef` позволяет получить идентификатор блока, предшествующего данному в цепи.

Функция `bcBlkIntegrityVerify` позволяет описать функцию, проверяющую целостность блока, в частности:

- Ассоциацию между заголовком и телом блока
- Ассоциацию между последовательными транзакциями в блоке
  - Полезно для описания взаимосвязи между различными транзакциями внутри блока. Пример взаимосвязи: всякая транзакия, переводящая средства с одного счёта на другой имеет в блоке соответствующую транзакцию, переводящую системе комиссия за выполненую операцию.

Функция `bcIsBetterThan` позволяет описать функцию выбора между двумя цепями изменений. Примером такой функции выбора является правило выбора цепочки с большей сложностью в Bitcoin.

Форком называется цепь, альтернативная принятой системой на текущий момент. Для любых двух цепей можно найти наименьшего общего предка lca (который в случае, если цепи не содержат ни одного совпадающего блока будет равен `Nothing`). Глубиной форка называют глубину общего предка lca, т.е. количество переходов, сделанных с помощью функции `bcPrevBlockRef`, начав с голоыного блока цепи, требующихся чтобы получить идентификатор lca.

Параметр `bcMaxForkDepth` позволяет лимитировать глубину форка. Любой форк, глубина которого превышает `bcMaxForkDepth`, рассматривается как некорректный. Подобная проверка необходима для реализации некоторых консенсус-алгоритмов, в частности требуется алгоритмом Ouroboros [14].

2.2.2. Применение цепи блоков

Помимо валидации цепи блоков, логика обработки блоков включает себя логику применения блоков к текущему состоянию сети. С точки зрения обработки блоков, состояние системы состоит из эффективного состояния системы `state` и хранилища блоков. Взаимодействие с состоянием системы описано конфигурацией, представленной в листинге 2.10.

```haskell
-- | Blund: BLock + UNDo
data Blund header payload undo = Blund
  { bwuBlock :: Block header payload
  , bwuUndo :: undo
  }

data BlkStateConfiguration header payload undo blockRef e m =
  BlkStateConfiguration
    { bsfApplyPayload :: payload -> ExceptT e m undo
    , bsfApplyUndo :: undo -> ExceptT e m ()

    , bsfStoreBlund :: Blund header payload undo -> m ()
    , bsfGetBlund :: blockRef -> ExceptT e m (Maybe (Blund header payload undo))
    , bsfBlockExists :: blockRef -> m Bool
    , bsfGetTip :: m (Maybe blockRef)
    , bsfSetTip :: Maybe blockRef -> m ()

    , bsfConfig :: BlkConfiguration header payload blockRef e
    }
```

Листинг 2.10 — Конфигурация обработки блоков

Методы конфигурации `BlkStateConfiguration` запускаются в некоторой монаде (окружении) `m` (монада – базовая конструкция, используемая во многих функциональных языках программирования, в том числе Haskell, для описания действий, выполняющихся последовательно в некотором окружении).

Методы `bsfApplyPayload`, `bsfApplyUndo` используются для работы с эффективным состоянием системы. Метод `bsfApplyPayload` применяет тело блока к состоянию, возвращая некоторый объект `undo`, который в дальнейшем может использоваться для отката изменений. Метод `bsfApplyUndo` применяет объект `undo` к состоянию, откатывая изменения, внесенные ранее некоторым `bsfApplyPayload`.

Методы `bsfStoreBlund`, `bsfGetBlund`, `bsfBlockExists`, `bsfGetTip`, `bsfSetTip` используются для взаимодействия с хранилищем блоков. Из сигнатур можно вывести что хранилище блоков должно содержать как минимум следующие данные:

– Идентификатор последнего примененного блока, `tip`
– Множество объектов типа `Blund`, представляющих из себя пару из примененного блока и соответствующего ему объекта `undo`

Таким образом, полное состояние системы можно представить как $s \in S_n$ для $n \in N \cup 0$:

$$
\begin{aligned}
&S_0 = \{\langle s_0 \in State, s_0 \in State, \emptyset, \bot \rangle\} \\
&S_{n \geqslant 1} = \{\langle s_0, s_n, blunds, tip \rangle \mid s_0, ..., s_n \in State, \\
&\quad blunds \in List_{Block \times Undo},\ tip \in Ref_{Block}, \\
&\quad (\ \exists \langle block, \_ \rangle \in blunds : ref(block) = tip\ ), \\
&\quad (\ \exists \langle s_0, s_{n-1}, blunds \setminus \{block\}, tip' \rangle \in S_{n-1} : \\
&\qquad \langle block, undo \rangle \in blunds \wedge\ tip' = prev(block) \\
&\qquad\quad \wedge s_{n-1} = applyUndo(undo, s_n), \\
&\qquad\quad \wedge s_n = applyPayload(payload(block), s_{n-1}), \\
&\quad), \\
&\}
\end{aligned}
\tag{2.2}
$$

2.2.3. Взаимосвязь с моделью состояния системы

В разделе 2.1 введена модель состояния системы. Модель блокчейна, описанная выше, однако, была потроена независимо. В настоящем подразделе мы покажем как модель блокчейна совмещается с моделью состояния системы.

Как показано выше, логика обработки блоков обращается с некоторым абстрактным состоянием `state`, к которому применяется `payload`, в результате чего возвращается объект `undo`, позволяющий откатить произведенные изменения. Абстрактные типы `state`, `payload`, `undo` введены намерено для того, чтобы лучше декомпозировать задачу валидации и применения блока на валидацию цепочки блока, оперирующую с объектом типа `header` и не рассматривающей `payload` и задачу валидации и применения `payload` к состоянию системы.

Для совмещения моделей требуется использовать подстановку типов: `payload = [StateTx id proof value]`, `undo = ChangeSet id value`. Реализация метода `bsfApplyPayload` последовательно валидирует и применяет каждую транзакцию из `payload` к состоянию (причем делает это таким образом, что в случае когда валидация очередной транзакции завершается неудачей, все примененные изменения откатываются). Реализация метода `bsfApplyUndo` применяет данный `undo` к состоянию.

Удобством такого разделения моделей блокчейна и состояния системы является то, что всякая компонента блокчейн-системы может быть представлена объектом `Validator e id proof value`, и, может быть, `BlockIntegrityVerifier header payload e`, которые проверяют требуемые инварианты для релевантных типов транзакций. Как можно заметить, тип `BlockIntegrityVerifier` также является моноидом, что позволяет композировать логику различных компонент системы моноидным умножением (что безусловно является значительным удобством при построении систем, позволяя реализовать плоскую, хорошо декомпозированную архитектуру для блокчейн-системы).

2.3. Расширение транзакции

2.4. Доступ к параметрам ОС

Резюме

ГЛАВА 3. МОДЕЛЬ БЛОКЧЕЙН-СИСТЕМЫ, ИСПОЛЬЗУЮЩЕЙ МЕТОД ДОКАЗАТЕЛЬСТВА ДОЛИ ВЛАДЕНИЯ

3.1. Обобщенный функционал метода доказательства доли владения

3.2. Делегация доли владения

3.3. Система обновления

Резюме

ГЛАВА 4. **МОДЕЛЬ КРИПТОВЛЮТЫ**

4.1. Аккаунтинг

4.2. Поддержка умных контрактов

4.3. Контракты с локальным состоянием

Резюме

ГЛАВА 5. **РЕАЛИЗАЦИЯ**

5.1. Реализация криптовалюты Cardano SL

5.2. Формализация модели на Haskell

Резюме

ЗАКЛЮЧЕНИЕ

TODO Заключение

СПИСОК ИСТОЧНИКОВ

1. Bitcoin blockchain. [Электронный ресурс]. URL: https://bitcoin.org.
2. NXT blockchain. [Электронный ресурс]. URL: https://nxtplatform.org.
3. Cardano blockchain. [Электронный ресурс]. URL: https://www.cardano.org/en/home.
4. Ethereum blockchain. [Электронный ресурс]. URL: https://ethereum.org.
5. NEO blockchain. [Электронный ресурс]. URL: https://neo.org.
6. Namecoin blockchain. [Электронный ресурс]. URL: https://namecoin.org.
7. Disciplina blockchain. [Электронный ресурс]. URL: https://disciplina.io.
8. Lamport Leslie [и др.]. Paxos made simple // ACM Sigact News. 2001. Т. 32, № 4. С. 18–25.
9. Ongaro Diego, Ousterhout John K. In search of an understandable consensus algorithm. // USENIX Annual Technical Conference. 2014. С. 305–319.
10. Nakamoto Satoshi. Bitcoin: A peer-to-peer electronic cash system. 2008.
11. Garay Juan, Kiayias Aggelos, Leonardos Nikos. The bitcoin backbone protocol: Analysis and applications // Annual International Conference on the Theory and Applications of Cryptographic Techniques / Springer. 2015. С. 281–310.
12. ZCash blockchain. [Электронный ресурс]. URL: https://z.cash.
13. IOTA blockchain. [Электронный ресурс]. URL: https://iota.org.
14. Ouroboros: A provably secure proof-of-stake blockchain protocol / Aggelos Kiayias, Alexander Russell, Bernardo David [и др.] // Annual International Cryptology Conference / Springer. 2017. С. 357–388.
15. Bentov Iddo, Pass Rafael, Shi Elaine. Snow White: Provably Secure Proofs of Stake. // IACR Cryptology ePrint Archive. 2016. Т. 2016. с. 919.
16. Micali Silvio. Algorand: The efficient and democratic ledger // arXiv preprint arXiv:1607.01341. 2016.

17. Miller Andrew, Xia Yu, Croman Kyle [и др.]. The Honey Badger of BFT Protocols. Cryptology ePrint Archive, Report 2016/199. 2016. https://eprint.iacr.org/2016/199.
18. Pedersen Torben Pryds. A Threshold Cryptosystem without a Trusted Party // Advances in Cryptology — EUROCRYPT '91 / под ред. Donald W. Davies. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991. С. 522–526.
19. Hyperledger: permissioned blockchain framework. [Электронный ресурс]. URL: https://www.hyperledger.org.
20. Danezis George, Meiklejohn Sarah. Centrally Banked Cryptocurrencies // CoRR. 2015. Т. abs/1505.06895. URL: http://arxiv.org/abs/1505.06895.
21. Maymounkov Petar, Mazieres David. Kademlia: A peer-to-peer information system based on the xor metric // International Workshop on Peer-to-Peer Systems / Springer. 2002. С. 53–65.
22. NEM Technical reference. [Электронный ресурс]. URL: https://nem.io/wp-content/themes/nem/files/NEM_techRef.pdf.
23. Goodman LM. Tezos: A self-amending crypto-ledger position paper. [Электронный ресурс]. 2014. URL: https://www.tezos.com/static/papers/white_paper.pdf.
24. Goldwasser Shafi, Micali Silvio, Rackoff Charles. The knowledge complexity of interactive proof systems // SIAM Journal on computing. 1989. Т. 18, № 1. С. 186–208.
25. Wood Gavin. Ethereum: A secure decentralised generalised transaction ledger. 2014. URL: https://ethereum.github.io/yellowpaper/paper.pdf.
26. Bitcoin developer documentation. [Электронный ресурс]. URL: https://bitcoin.org/en/developer-documentation.
27. EOS Technical whitepaper. [Электронный ресурс]. URL: https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md.
28. Peercoin whitepaper. [Электронный ресурс]. URL: https://peercoin.net/assets/paper/peercoin-paper.pdf.
29. Bentov Iddo, Gabizon Ariel, Mizrahi Alex. Cryptocurrencies Without Proof of Work // Financial Cryptography and Data Security / под ред. Jeremy Clark,

Sarah Meiklejohn, Peter Y.A. Ryan [и др.]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. С. 142–157.

30. Chepurnoy Alex. Scorex 2.0 framework. [Электронный ресурс]. 2016. URL: https://github.com/ScorexFoundation/Scorex.
31. Chepurnoy Alex. Scorex 2.0 framework tutorial. [Электронный ресурс]. 2016. URL: https://github.com/ScorexFoundation/ScorexTutorial/blob/master/scorex.pdf.
32. Free haskell package. [Электронный ресурс]. URL: https://hackage.haskell.org/package/free.