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.

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

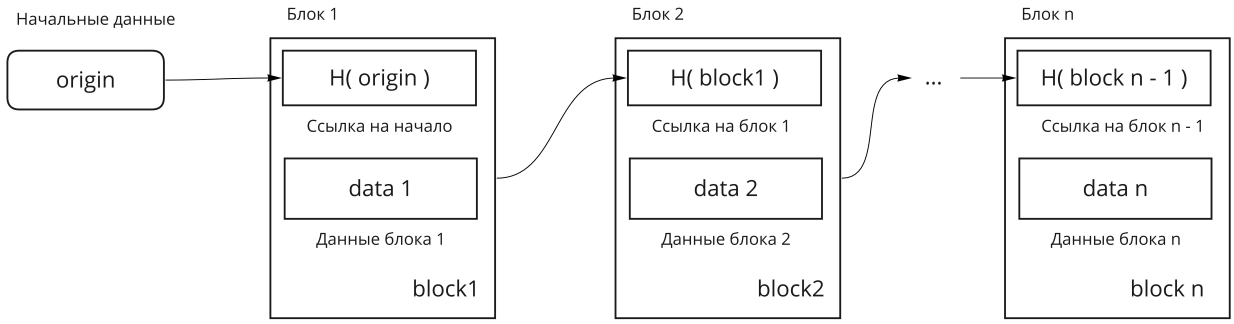

Рис. 1.1 — Диаграма структуры Блокчейн

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 1.2. 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.

1.6. Система типов и синтаксис языка Haskell

TODO описать что используется Haskell версии GHC X.X, что под капотом там такаято система и почему система типов подходит для описания модели криптовалюты; описать основные синтактические конструкции которые в продолжение повествования будут использоваться

TODO типы: Word32, Rec, Semigroup, COnst, NotIntersects/HasCap HasPrism, HasException, Proxy

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 1.1 — 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.

```
type family RecAll' (rs :: [u]) (c :: u -> Constraint) :: Constraint where
   RecAll' '[] c = ()
   RecAll' (r ': rs) c = (c r, RecAll' rs c)
```

Листинг 1.2 — Семейство типов $RecAll$

TODO: что такое тип, вид (как эти понятия используются)

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 1.3 — 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.

1.6.1. Библиотека Vinyl

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

Модель блокчейн-системы определяет блокейн-систему общего вида, т.е. систему обработки данных, использующую структуру блокчейн.

Модель блокчейн-системы вводится как набор следующих компонент:

– модель локального состояния системы;
– модель обработки блоков;
– механизм расширения транзакций;
– механизм обращения к изменяющимся параметрам ОС.

Модель локального состояния системы определяет вид хранилища данных, используемого системой, способы взаимодействия с хранилищем (чтения и записи данных). TODO описать остальные компоненты после описания chapter'a

2.1. Модель локального состояния системы

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

2.1.1. Тип состояния системы

Тип `StateP1`, представленный в листинге 2.1, выражает тип состояния системы. Причем как глобального состояния целиком, так и его части. Состояние представляется как словарь, отображение из ключа `key` в значение `value`. И ключ, и значение введены как абстрактные типы. Это позволяет использовать определения для произвольных ключей и значений, используя для разных типов одни и те же полиморфные функции.

```
1 import Data.Map (Map)
2
3 type StateP1 key value = Map key value -- Portion of state
```

Листинг 2.1 — Тип состояния системы 'StateP1'

Тип `StateP1` описывает состояние, в котором типы ключа и значения фиксированы и одинаковы для всех компонент одной системы. Это в принципе

позволяет описать многокомпонентную систему, даже учитывая различия репрезентации данных в различных компонентов. В частности возможно задать типы ключа и значения как суммы частных типов конкретных компонент, как продемонстрировано в листинге 2.2. Следует также заметить, что функциям, работающими с данными каких-либо отдельных компонент системы, будет требоваться выбирать из хранилища ключи, созданные интересующим конструктором и проверить соответствие конструктора значения (потенциально завершая исполнение исключением).

```
data Address = ..
data PublicKey = ..
data Signature = ..

data ExampleKey1 = EkDlg PublicKey | EkBalances Address

data ExampleValue1 = EvDlg (Address, Signature) | EvBalances Word32
```

Листинг 2.2 — Пример использования структуры состояния: тип-сумма

Для написания функций, работающих с отдельными компонентами системы было бы удобно разделять состояние системы на несколько частей, задавая типы для каждой из частей отдельно. Это сделать можно, представляя состояние как тип-произведение нескольких состояний `StateP1` (см. листинг 2.3), в частности для системы из двух компонент в виже пары. Однако такое представление имеет существенный минус: требуется использовать конкретные типы, либо зависеть от количества компонент, что затрудняет реализацию полиморфных функций, работающих с состоянием системы в целом (не частных компонент).

```
type StateP2 =
  ( StateP1 PublicKey (Address, Signature)
  , StateP1 Address Word32
  )
```

Листинг 2.3 — Пример использования структуры состояния: тип-произведение

Кроме того, для формализации дополнения транзакций (см. раздел 2.3) требуется указывать в типе информацию о том, данные каких именно компонент содержаться в объекте состояния. Для разрешения этих затруднений окончательный тип состояния системы `StateP` получился несколько более сложным и для его определения используется ряд продвинутых возмож-

ностей системы типов Haskell (в частности семейства типов, поднятие типов до видов).

```haskell
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE TypeInType #-}

import Data.Vynil
import Contol.Lens (iso, Iso')

type StateP (mw :: ((* -> *) -> u -> *)) (rs :: [u]) = Rec (mw Identity) rs

class (Show (K mw t)) =>
  WrappedM (mw :: ((* -> *) -> u -> *)) (t :: u) where
    type K mw t :: *
    type V mw t :: *
    wrappedM' :: forall (g :: * -> *) . Iso' (mw g t) (U g mw t)
    consA :: forall (g :: * -> *) . U g mw t -> mw g t

type U g mw t = Map (K mw t) (g (V mw t))
```

Листинг 2.4 — Тип состояния системы $StateP$

В определении типа `StateP`, представленном в листинге 2.4, используется тип `Rec` из библиотеки `vinyl` (см. подраздел 1.6.1). Он позволяет задать гетерогенный список хранилищ различных компонент. Тип `StateP` полагается на существование вида `u`. Произвольный тип `r`, принадлежащие виду `u`, идентифицирует некую компоненту. Тип `mw` параметризуется типом `g`, в который оборачивается тип-значение, и типом `r`, возвращая тип хранилища для соответствующей `r` компоненте, причем возвращаемый тип изоморфен `Map k (g v)` для некоторых `k`, `v` (соответствующих `r`). Изоморфизм типу `Map k (g v)` задается инстансом класса `WrappedM`.

Для внесения большей ясности в природу типа `StateP` и его параметров, рассмотрим пример использования, представленный в листинге 2.5. Тип данных `ExampleStorage` при включенном расширении `DataKinds` порождает вид `ExampleStorage` с двумя возможными типами: `ESDlg`, `ESBalances`. Семейство типов `EST` задаёт типы ключа и значения. Тип `M` принимает тип-обертку значения `g :: *-> *` и тип-маркер компоненты `f :: ExampleStorage` и является оберткой над ассоциативным массивом, представляющим состояние (соответствующий инстанс класса `WrappedM` позволяет получить доступ к объекту типа `Map`). Значение `exStateP` содержит объект состояния, в котором содержатся ключи компоненты `ESDlg`, в то время как `exStateP2` содержит ключи компонент `ESBalances`, `ESDlg`. Следует заметить, что вследствие того что типы хранилищ

```haskell
{-# LANGUAGE DataKinds #-}

type family Fst a where Fst '(x,y) = x
type family Snd a where Snd '(x,y) = y

data ExampleStorage = ESBalances | ESDlg
  deriving Show

type family EST (f :: ExampleStorage) :: (*, *) where
  EST 'ESBalances = '(Int, Word16)
  EST 'ESDlg = '(Int, String)

newtype M g f = M { unM :: Map (Fst (EST f)) (g (Snd (EST f))) }

instance (Show (Fst (EST t))) => WrappedM M t where
    type K M t = Fst (EST t)
    type V M t = Snd (EST t)
    wrappedM' = iso unM M
    consA = M

exStateP :: StateP M '[ 'ESDlg]
exStateP = M (M.fromList [ (30, Identity "someValue") ]) :& RNil

exStateP2 :: StateP M '[ 'ESBalances, 'ESDlg]
exStateP2 =
  M (M.fromList [ (30, Identity 1005) ])
    :& M (M.fromList [ (30, Identity "someValue") ]) :& RNil
```

Листинг 2.5 — Пример использования типа $StateP$

задаются семейством типов `EST`, использование неподходящего типа значения приведет к ошибке компиляции.

2.1.2. Тип транзакции к состоянию

Рассмотрим изменение конкретного ключа в состоянии. В листинге 2.6 представлен тип `ValueOp`, представляющий из себя объект-изменение ключа в состоянии. Тип `ValueOp` задает несколько конструкторов. Конструктор `New` создаёт новое значение, `Rem` – удаляет существующее значение. Конструктор `Upd` обновляет существующее значение (возможно, возвращая ошибку), конструктор `NotExisted` является маркером того что значение не существует (тип `NotExisted` эквивалентен последовательному применению `New x`, `Rem`). Следует заметить, что для типа `ValueOp` нельзя определить композицию, т.к. последовательное применение некоторых конструкторов приводит к ошибке (например, `Rem` и `Upd`). Поэтому задаётся тип `ValueOpEx`, для которого реализуем инстанс `Semigroup`.

В листинге 2.7 вводится тип изменения состояния. В объекте изменения состояния для каждой компоненты состояние определяется объект ти-

```haskell
data ValueOp v
   = New v
   | Upd (v -> v)
   | Rem
   | NotExisted
   deriving (Show, Eq)

data ValueOpEx v
   = Op (ValueOp v)
   | Err
   deriving (Show, Eq)

instance Semigroup (ValueOpEx v) where
  (<>) a b = ...
```

Листинг 2.6 — Изменение значения некоторого ключа

па `Map k (ValueOp v)`, т.е. для некоторого подмножества ключей состояния мы задаём способ их изменения. Требуется заметить, что применение объекта изменений может завершиться ошибкой (например, если ключу, отсутствующему в состоянии, сопоставлено изменение `Rem`). В листинге 2.7 также представлен тип функции `chgsetMappend`, с помощью которой можно композировать последовательные изменения состояния. Семейства типов `RecAll`, `RecAll'` используются для установления ограничения на тип для каждого $r \in rs$.

```haskell
type ChangeSet (mw :: ((* -> *) -> u -> *)) (rs :: [u]) = Rec (mw ValueOp) rs

chgsetMappend
  :: forall rs mw .
    ( RecAll' rs (WrappedM mw)
    , Semigroup (Rec (mw ValueOpEx) rs)
    )
  => ChangeSet mw rs -> ChangeSet mw rs -> VerRes String (ChangeSet mw rs)
```

Листинг 2.7 — Изменение состояния системы

Транзакция к состоянию представлена в листинге 2.8. 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. Поля `txType`, `txProof` 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.

```
newtype StateTxType = StateTxType Int

data StateTx mw rs proof = StateTx
   { txType :: StateTxType
   , txProof :: proof
   , txBody :: ChangeSet mw rs
   }
```

Листинг 2.8 — Транзакция

2.1.3. Вычисление в модели состояния

Опр. 2.1. Вычисление в модели состояния – функция, выполняющая последовательность запросов к состоянию для возврата результата.

Задачей настоящего подразадела является формулировка типа вычисления в модели состояния. При построении типа используются следующие соображения:

– вычисление может выполнить произвольное количество запросов к состоянию, причем содержимое этих запросов, равно и их количество зависят от результатов предыдущих запросов;
– вычисление не имеет возможности записи в состояние системы, только чтения.

В листинге 2.9 представлен тип запроса множества ключей из состояния `StateReq`. Как можно заметить, его конструкция практически идентична конструкциям типов `StateP` и `ChangeSet`, с отличием в том, что в качестве обертки значения используется тип `Const ()`, что в результате означает что значения хранится не будут и ассоциативный массив используется как множество ключей.

```
type StateReq (mw :: ((* -> *) -> u -> *)) (rs :: [u])
   = Rec (mw (Const ())) rs
```

Листинг 2.9 — Запрос множества ключей из состояния

Тип `StateAccess1`, представленный в листинге 2.10, представляет из себя пару, первым элементом которой является запрос к состоянию, вторым – функция, которая принимает результат исполнения запроса и возвращает

функцию продолжения вычисления. В некоторых случаях это может оказаться недостаточным, например если требуется запросить не какой-то конкретный ключ, а все ключи, отвечающие какому-либо критерию.

В простейшем случае требуется поддержка итерации всех ключей части состояния, соответствующей некоторой компоненте, что и реализовано в типе `StateAccess`, который будет использован в дальнейшем. Для этого вводится тип `StateIter`, аналогичный `StateReq`, построенный на основе типа `FoldF res a` – функции итерации по ключам типа `a`, возвращающей значение res. На основе `StateIter` вводится функтор доступа к состоянию `StateAccess`.

```
newtype StateAccess1 res mw rs
   = StateAccess1 (StateReq mw rs, StateP mw rs -> res)


data FoldF a res = forall b. FoldF (b, a -> b -> b, b -> res)
newtype MFoldF res a = MFoldF { unMFoldF :: Maybe (FoldF a res) }

newtype StateIter (mw :: ((* -> *) -> u -> *)) (rs :: [u]) (res :: *)
   = StateIter { unStateIter :: Rec (mw (MFoldF res)) rs }

instance Functor (FoldF a) where (..)
instance RecAll' rs (WrappedM mw) => Functor (StateIter mw rs) where (..)

data StateAccess mw rs res =
  StateAccess
    { sQuery :: StateReq mw rs
    , sHandler :: StateP mw rs -> res
    , sIterator :: StateIter mw rs res
    }
  deriving Functor
```

Листинг 2.10 — Функтор доступа к состоянию

Функтор доступа к состоянию `StateAccess` позволяет либо однократно запросить произвольное множество ключей из состояние, либо однократно проитерироваться по всем ключам выбранных компонент состояния. Однако для многократного обращения к состоянию в качестве переменной типа `res` требуется подставить такой тип, который бы позволял повторить запрос неограниченное количество раз.

***Свойство* 2.1.** *Функтор доступа к состоянию $StateAccessmwrs$, примененный к типу с моноидальной структурой res, является моноидом.*

Свойство 2.1 показывается довольно просто (это следует из моноидальной структуры типа `Rec`) и будет использовано в дальнейшем.

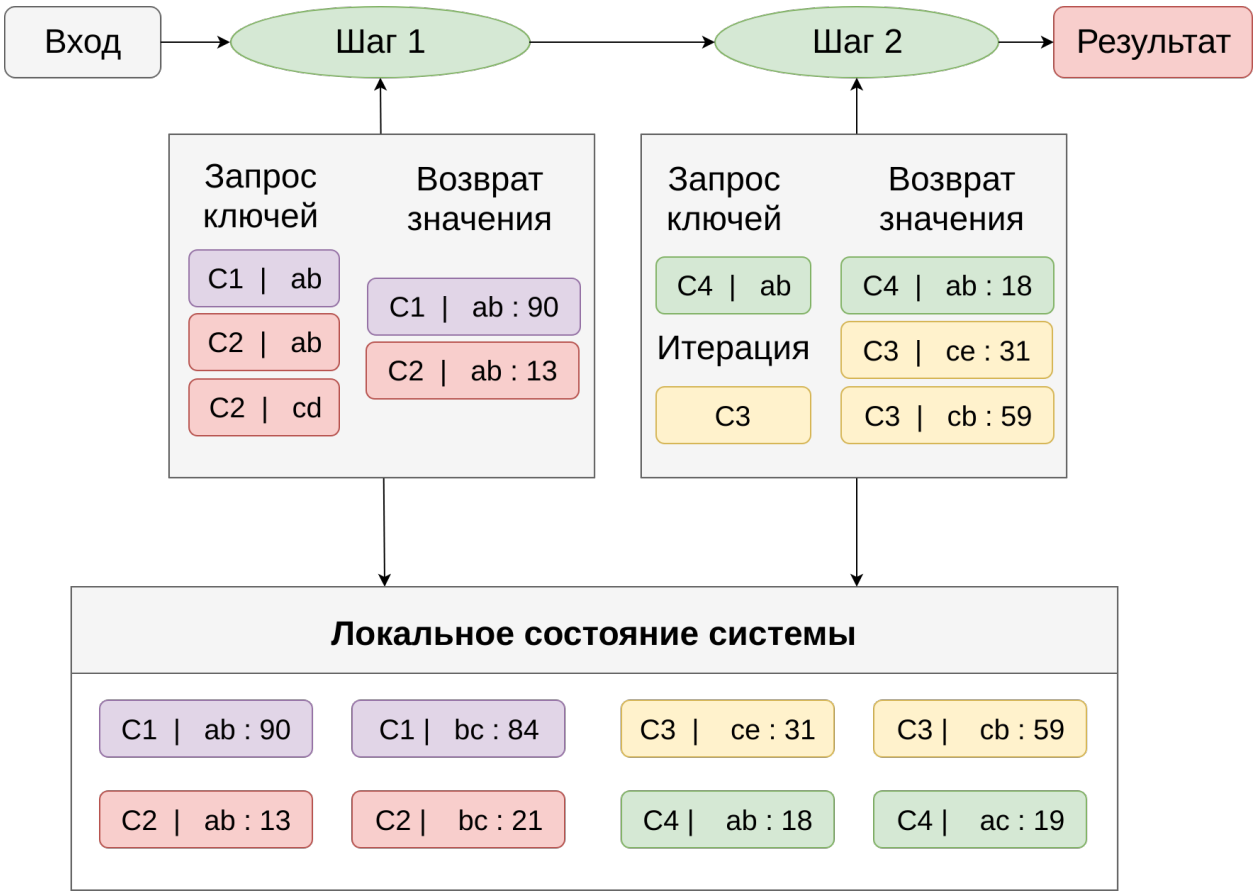

Рис. 2.1 — Пример исполнения вычисления с многократным доступом к состоянию

Существует несколько распространенных методов для моделирования вычисления с многократным обращением к состоянию в Haskell. Наиболее распространенным является моделирование вычислений с помощью классов типов, в частности используя стандартный класс `Monad`. В листинге 2.11 представлен тип `RoComp`. Как следует из его определение, вычисление обладающее типом `RoComp mw rs` (при наличие инстанса `WrappedM mw rs`) обладает следующими возможностями:

– последовательная композиция двух вычислений в случае когда одно вычисление зависит от результата предыдущего (возможность унаследована от класса `Monad`, метода `>>=`);
– параллельная композиция двух вычислений в случае когда вычисления не зависят от результата друг друга (метод `parallel` класса Effectful, доступный для типа `E` через оператор `<>`);
– чтение значений из состояния посредством запроса ключей или итерация по всем ключам состояния (метод `effect` вместе с предлставленым объектом функтора `StateAccess`).

```haskell
class Monad m => Effectful eff m | m -> eff where
   parallel :: Semigroup a => m a -> m a -> m a
   effect :: eff a -> m a

newtype Eff eff a = Eff { unEff :: forall m . Effectful eff m => m a }
  deriving Functor

instance Semigroup a => Semigroup (Eff eff a) where
   Eff a <> Eff b = Eff (parallel a b)

instance Applicative (Eff eff) where (..)
instance Monad (Eff eff) where (..)

execEff :: eff a -> Eff a
execEff e = Eff (effect e)

type RoComp mw rs = Eff (StateAccess mw rs)
type ERoComp e mw rs = ExceptT e (RoComp mw rs)
```

Листинг 2.11 — Тип вычисления, обращающегося к состоянию

Тип `ERoComp` представляет вычисление, которой к возможнностям вычисления типа `RoComp` добавляет возврат ошибки исполнения (в случае возникновения исключения). Возможность возврата ошибки является потребностью в выражении многих функциональностей.

Важно отметить, что вычисление `RoComp` имеет только возможность доступа к состоянию посредством функтора `StateAccess`, никакие другие сайд-эффекты ему недоступны. В частности вычисление `RoComp` не имеет доступа к:

– операциям с файловой системой;
– операциям с сетью;
– операциям с изменяемыми переменными (инстурументами синхронизации данных между потоками);
– операциям записи в состояние.

Такая ограниченность вычислений типа `RoComp` позволяет более четко рассуждать о произвольном вычислении такого типа, в том числе о его исполнения в отдельном потоке или композиции с другими вычислениями.

Следует отметить, что параллельную композицию можно реализовать с помощью последовательной, используя метод `>>` класса **`Monad`**. Однако введение параллельной композиции как отдельного метода класса `Effectful` позволяет получить конструкцию с интересными свойствами, которые будут рассмотрены в следующем подразделе.

2.1.4. Композиция вычислений в модели состояния

Рассмотрим следующую задачу: дано вычисление `comp :: a -> StateTx mw rs proof -> RoComp mw rs a`, требуется обобщить его на случай множества транзакций, т.е. реализовать функцию `compMany :: a -> [StateTx mw rs proof] -> RoComp mw rs [a]`. Есть несколько особенностей, на которые следует обратить внимание при построении функции `compMany`:

- в случае когда длина входного списка транзакций достаточно велика, а вычисление `comp` не является тривиальным, эффективная реализация `compMany` на многопроцессорной системе может требовать использования возможностей параллелизма (предоставляемых языком и системой);
- вычисление `comp` примененное ко второй транзакции из списка может ожидать в качестве первого параметра `a` как переданное изначально значение `a`, так и результат выполнения вычисления `comp` для первой транзакции.

Для того чтобы упростить отслеживание зависимостей между частями вычисления `compMany`, а также понимание возможности многопоточного исполнения вычисления, при формулировании класса типов `Effectful` были созданы возможности для последовательной и параллельной композиции с помощью операторов `>>=` и `<>` соответственно. С помощью выделения этих двух типов композиций для вычисления типа `RoComp`, применяемого к множеству входных данных, можно сформулировать одно важное свойство. Прежде чем перейти к формулировке свойства, введём понятие глубины вычисления.

Опр. 2.2. Глубина вычисления в модели состояния – количество обращений к состоянию, которое сделает вычисление при некотором входе, максимизированное по всевозможным входам и минимизированное по всевозможным стратегиям запуска вычисления.

Определение 2.2 можно также представить в виде уравнения:

$$\begin{aligned} &\text{depth}\ (\ comp \in RoComp\ ) \equiv \\ &\min_{Strategy}\ \max_{Input}\ \{\ \text{CountStateAccess}(\text{execute}\ (Strategy,\ comp,\ Input))\ \} \end{aligned} \tag{2.1}$$

Наибольший интерес представляют две стратегии вычисления:

– стратегия, при которой достигается минимум в формуле 2.1;
– стратегия, при которой возможно наибольшее распараллеливание вычислений.

Опр. 2.3. Стратегия с минимизацией запросов к состоянию – стратегия исполнения вычисления в модели состояния, при которой будет совершено минимальное количество запросов к состоянию.

Реализация стратегии из определения 2.5 возможна благодаря свойству 2.1 функтора доступа к состоянию (о возможности объединения любых двух запросов в один). В частности сформулируем стратегию S_1. Вычисление при использовании S_1 будет исполняться в один поток. Вычисление в моделе состояния состоит из запросов к состоянию, использования последовательной и параллельной композиции. Следует заметить, что в случае последовательной композиции двух запросов нет возможности объединить их в один запрос, т.к. последующий запрос непосредственно зависит от результата исполнения предыдущего.

Использование параллельной композиции двух вычислений позволяет оптимизировать количество запросов к состоянию. В стратегии S_1 происходит объединение запросов двух вычислений a и b, а именно: будут взяты первые запросы из вычислений a и b, объединены в один и исполнены, затем произведена последовательная композиция (в которую подставлен результат объединенного запроса) и взяты вторые запросы, объединены и исполнены и т.д. Т.к. использование последовательной композиции к последовательным запросам является неустранимым, полученная стратегия S_1 является стратегией с минимизацией запросов к состоянию.

Опр. 2.4. Стратегия с максимизацией параллелизма – стратегия исполнения вычисления в модели состояния, при которой при каждом использовании параллельной композиции запускается отдельный поток вычисления.

В примере на рисунке 2.2 показывается использование обоих стратегий для запуска трёх выислений A, B, C, причем вычисление B зависит от результата исполнения A. Т.о. композицию вычислений можно представить в виде формулы `(a >>= b)<> c`. Каждое из трех вычислений на рисунке рассмат-

ривается как последовательность запросов, пронумерованная натуральными числами.

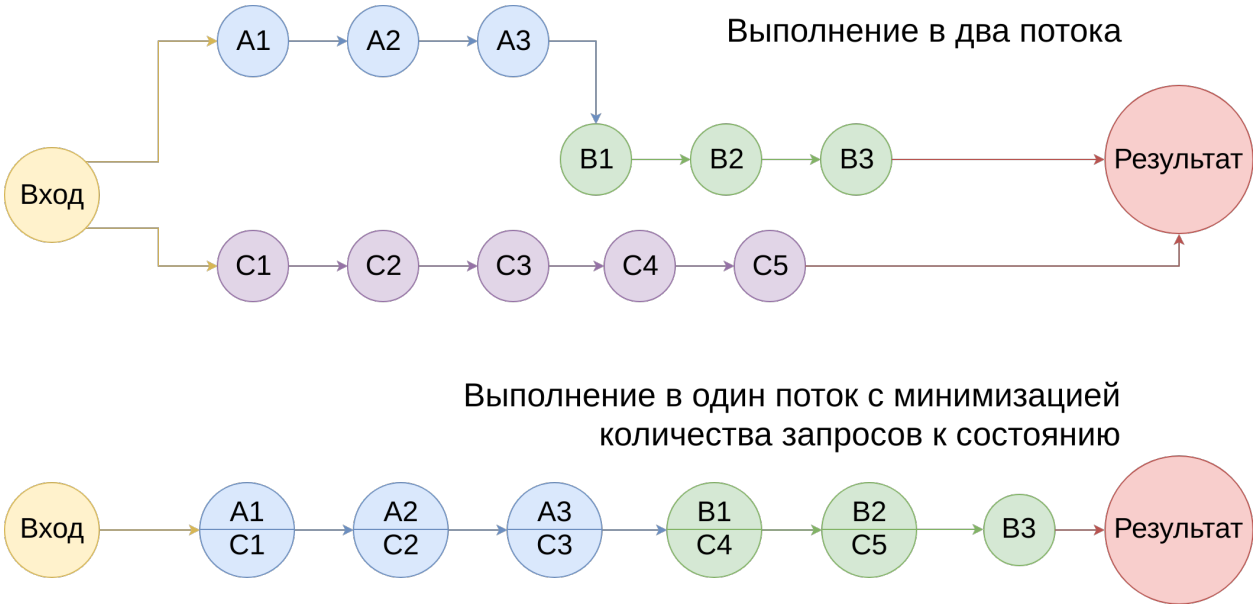

Рис. 2.2 — Две стратегии запуска трёх вычислений

Опр. 2.5. Композиция фискированной глубины – композиция произвольного числа вычислений в модели состояния, глубина которой не зависит от числа вычислений входящих в композицию.

***Лемма* 2.1.** *Композицию N вычислений фиксированной глубины, каждое из которых делает фиксированное количество запросов к состоянию, возможно выполнить в $O(N)$ потоков, каждый из которых сделает фиксированное количество запросов к состоянию.*

Доказательство: Воспользуемся стратегией с максимизацией параллелизма. Из фиксированной глубины композиции следует, что максимальная длина цепочки последовательных композиций также фиксирована. Поскольку каждое из N вычислений сделает фиксированное количество запросов к состоянию, в цепи последовательных композиций фиксированное число элементов, каждый запузенный поток также совершит фиксированное число запросов к состоянию. Новый поток мы будем запускать при каждом использовании параллельной композиции, каждое из N вычислений может содержать внутри использования параллельной композиции, однако не более фиксированного числа. Соответственно, будет запущено не более $O(N)$ потоков. □

2.1.5. Валидация транзакции

Простейший валидатор транзакции может быть описан функцией, имеющей одну из сигнатур, представленных в листинге 2.12.

```
validator1 :: StateTx mw rs proof -> Bool
validator1 = ...

validator2 :: StateP mw rs -> StateTx mw rs proof -> Bool
validator2 = ...

validator3 :: StateTx mw rs proof -> (StateReq mw rs, StateP mw rs -> Bool)
validator3 = ...

validator4 :: StateTx mw rs proof -> RoComp id value Bool
validator4 = ..
```

Листинг 2.12 — Примеры сигнатур валидатора

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 неэффективным 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 ми-

нимум `maxDlgHeight` запросов к базе, где `maxDlgHeight` – максимальная путь в ациклическом графе делегации (процесс валидации делегационной транзакции будет подробно рассмотрен в разделе 3.2).

Окончательный тип валидатора транзакции предложен в листинге 2.13:

```
newtype PreValidator e mw rs proof =
  PreValidator
    (StateTx mw rs proof -> ERoComp e mw rs ())

newtype Validator e mw rs proof =
  Validator (Map StateTxType (PreValidator e mw rs proof))
```

Листинг 2.13 — Тип валидатора транзакции

Тип `PreValidator` является достаточным для описания логики валидации, в частности при проверки 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`, содержащий 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`.

***Свойство* 2.2.** *Предвалидатор и валидатор являются моноидами.*

2.2. Модель обработки блоков

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 2.14 — 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 в разделе 2.1). Логика валидации цепи блоков, конфигурация которой представлена в листинге 2.15, 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 $c2$

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

newtype ChainComparator header =
     ChainComparator
       { runChainComparator
           :: OldestFirst [] header
           -> OldestFirst [] header
           -> Bool
       }

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

Листинг 2.15 — Конфигурация валидации цепи блоков

  - 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.

Опр. 2.6. Форк – 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 2.16.

```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 2.16 — 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 мо-

жет 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 из-

менения. 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 = [rawTx]`, `undo = ChangeSet mw rs`. Подробнее обработка транзакции, хранящейся в блоке, преобразование её в `StateTx mw rs proof`, будет рассмотрено в разделе 2.3.

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 mw rs proof`, 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. Механизм дополнения транзакции

В разделе 2.1 определен механизм валидации и применения изменений к состоянию системы. Следует заметить, что изменения вносятся посредством транзакций, представляющие (в соответствии с листингом 2.8) из себя объект-тройку из целочисленного значения, объекта доказательства `proof` и объекта изменений `ChangeSet mw rs`. При построении модели блокчейн-системы следует учесть, что коммуникация между узлами сети происходит посредством байт, а не этих абстрактных объектов.

Процесс преобразования объектов, хранящихся в оперативной памяти в объекты, передаваемые по сети, принято обозначать термином десериализация. Под сериализацией зачастую подразумевается биективное преобразование между некоторым множеством объектов в оперативной памяти и последовтаельностями байт. Тривиальный случай десериализации – когда биективное преобразование может быть реализовано с помощью чистой функции (не требущей доступа к какому-либо контексту). Однако в общем случае функция десериализации может требовать доступа к состоянию, в частности представление изменения состояния может быть закодированно компактным способом, требующим получения значений из состояния для однозначного декодирования.

Моделировать этот процесс возможно следующим образом: сериализацию следует проводить в два этапа. Первый этап десериализации проходит без доступа к состоянию, формирует объект доказательства и часть объекта изменений состояния. Второй этап имеет доступ к состоянию, дополняя набор изменений состояния новыми парами (ключ – изменение значения). В рамках модели мы рассмотрим именно второй этап, здесь и далее называемый дополнением транзакции.

2.3.1. Дополнение транзакции общего вида

Функцию дополнения транзакции можно представить в виде типа `Expander1`. Для десериализации транзакции тогда потребуется конфигурация `Expander1Conf`.

```
data Expander1 rawTx e mw rs = Expander1
   { expander1Act :: rawTx -> ERoComp e mw rs (ChangeSet mw rs)
   }

data Expander1Conf rawTx e mw rs proof = Expander1Conf
   { e1cExpander :: Expander1 rawTx e mw rs
   , e1cDeserialize :: ByteString -> Either e rawTx
   , e1cTypeProof :: rawTx -> (StateTxType, proof)
   }
```

Листинг 2.17 — Дополнение транзакции общего вида

Тип `Expander1` описывает вычисление в модели состояния. Естественным образом встаёт вопрос о существовании композиции фиксированной глубины для дополнения множества транзакций. В общем случае такой композиции не существует. Вычисление дополнения для очередной транзакции

k зависит от состояния, которое будет получено как результат применения предшествующих транзакций $1, .., k-1$ (см. диаграмму на рисунке **??**). Как следствие, требуется применить последовательную композицию как минимум $k-1$ раз, что означает что глубина композиции не является фиксированной.

2.3.2. Дополнение транзакции с ограничением

Фиксированная глубина для композиции вычислений дополнения произвольного множества транзакций, однако, достижима при условии наложения ограничений на вид функции дополнения транзакции. Для получения композиции фиксированной глубины требуется убрать зависимость дополнения транзакции k от результата дополнений предшествующих транзакций $1, .., k-1$.

В листинге 2.18 вводится новый тип дополнения транзакции `Expander`, который позволяет разграничить компоненты состояния, из которых производится чтения от компонент состояния, значения из которых фигурируют в результирующем объекте изменения состояния. Сигнатура и реализация функции `applyExpanderTxsPar` используют идею дополнения транзакции с ограничением: ограничение типа `NotIntersects` гарантирует что компоненты состояния-выход функции дополнения не пересекаются с компонентами состояниями, из которых функция дополнения производит чтение. Реализация функции `applyExpanderTxsPar` оказывается чрезвычайно простой: с помощью вспомогательной функции для работы с моноидами `mconcat` мы применяем параллельную композицию для функции дополнения, соответственно примененной к каждой транзакции из списка. Глубина полученной композиции не зависит от числа транзакций.

```
1  newtype Expander rawTx e mw inRs outRs =
2    Expander
3      { runExpander :: rawTx -> ERoComp e mw inRs (ChangeSet mw outRs) }
4
5  applyExpanderTxsPar
6     :: NotIntersects inRs outRs
7     => Expander rawTx e mw inRs outRs
8     -> [rawTx]
9     -> ERoComp e mw inRs [ChangeSet mw outRs]
10 applyExpanderTxsPar (Expander runExpander)
11          = mconcat . map (fmap pure . runExpander)
```

Листинг 2.18 — Дополнение транзакции с ограничением

Однако дополнения транзакции такого вида может быть недостаточно. В частности, может понадобиться произвести процедуру дополнения в несколько этапах, на каждом применяя свою функцию и применяя результаты дополнения предыдущего этапе к состоянию, используемому в настоящем. Это можно представить как последовательность функций дополнения, применяемых к списку транзакций, причем сперва первая функция дополнения применяется ко всем транзакциям из списка, затем вторая и т.д.

В случае использования нескольких последовательных функций дополнения es формула ограничения несколько усложняется:

$$\forall e_i \in es \;\; \forall e_{j \geqslant i} \in es \quad in\ (e_i) \cap out\ (e_j) = \emptyset \tag{2.3}$$

При условии выполнения формулы 2.3 (соответствующее ограничение задаётся аналогично ограничению в листинге 2.18) использование нескольких последовательных функций дополнения по-прежнему позволяет построить композицию фиксированной глубины. На рисунке 2.3 представлена схема выполнения такой композиции (результатам дополнения соответствует список из объектов типа `ChangeSet`, получающийся в результате выполнения всех функций дополнения).

2.4. Доступ к динамическим переменным окружения

В некоторых случаях, в частности при построении большинства алгоритмов консенсуса для построения обработчика приходящих транзакций и блоков может потребоваться доступ к некоторым изменяющимся параметрам окружения. В качестве такого параметра может выступить любой параметр, о котором можно выдвинуть предположении о синхронизированности между множеством узлов сети. Единственным используемым на практике параметром, однако, является локальное время (синхронизация локального времени между множеством узлов сети возможна в случае если на всех машинах включен сервис NTP или аналогичные ему).

Для предоставления доступа к динамическим параметрам окружения в предлагаемой модели блокчейн-системы предлагается воспользоваться мо-

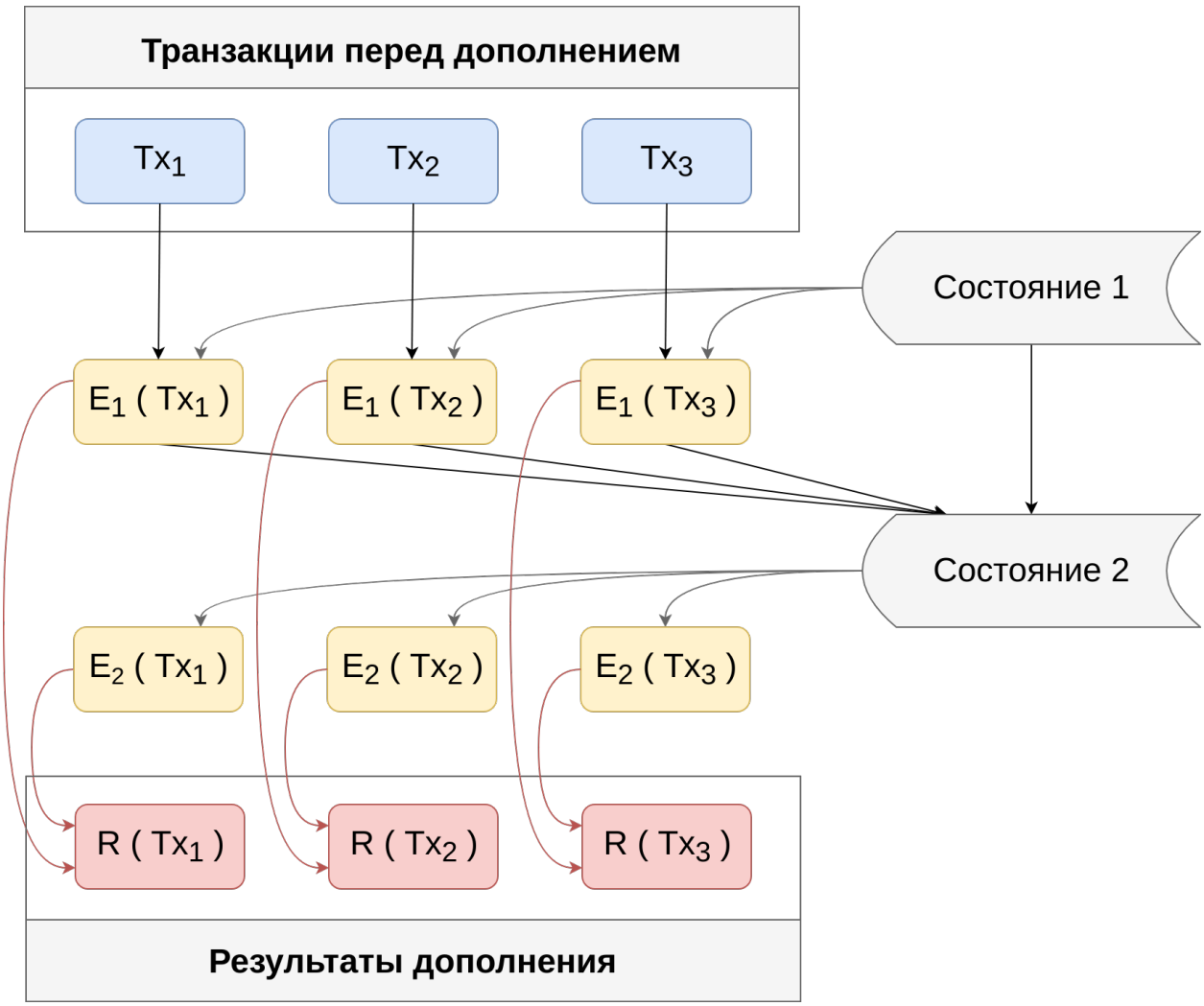

Рис. 2.3 — Применение последовательных функций дополнения E_1, E_2 к списку транзакций

делью состояния: специальный компонент в модели состояния будет содержать отображение из имени параметра в его последнее значение. Таким образом любое вычисление в модели состояния будет иметь доступ к любому из изменяющихся параметров.

Однако для завершения построения требуется рассмотреть функционал, который будет обновлять значения в состоянии. В рассмотренных ранее частях модели блокейн-системы, изменение состояние производилось только функционалом обработки цепи блоков (раздел 2.2), теперь же помимо этого функционала должен существовать некий гипервизор (от англ. hypervisor – надсмотрщик), следящий за изменением параметров окружения и обновляющий состояние соответствующим образом.

Кроме того, следует заметить, что корректность многих операций при вычислениях в модели состояния напрямую зависит отсутствия возможности

записи в вычислении модели состояния и, вообще говоря, от неизменности состояния во время проведения операции (применения блоков, в частности). Вследствии этого гипервизор должен быть реализован в связке с функционалом обработки блоков, чтобы упорядочить события обработки блоков и изменения параметров окружения.

Резюме

TODO резюме

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

Метод доказательства доли владения (или метод PoS) является одним из наиболее распространенных методов построения алгоритмов консенсуса в блокчейн-системах и был подробно рассмотрен в подразделе 1.2.2 главы 1 настоящей работы.

В настоящей главе будет рассмотрена модель блокчейн-системы, использующей метод доказательства доли владения. Будет введен обобщенный функционал метода доказательства доли владения, с его использованием описаны функционалы делегации и системы обновления.

3.1. Обобщенный функционал метода доказательства доли владения

Блокчейн-система, использующая метод доказательства доли владения, обладает следующими характеристиками:

– система хранит информацию о долях владения;
– всякий блок сопровожден подписью его создателя;
– право создания блока определяется по идентификатору создателя, текущему времени и, возможно, некоему объекту доказательства, хранящемуся в одной из транзакций блока.

На основании этих наблюдений можно сформулировать обобщенный функционал метода делегации доли владения. В листинге 3.1 представлена конфигурация обобщенного функционала `StakeConfiguration`. Метод `scGetStake` позволяет получить значение доли владения некоторого участника. Этот метод будет в дальнейшем использован для построения функциональностей, использующих информацию о долях владения. Метод `scEligibleToForge` позволяет по идентификатору создателя блока `stId` и некоторому объекту доказательства `proof`, проверить что `stId` имел право выпустить блок при текущем состоянии системы (метод `scEligibleToForge` завершится ошибкой в случае, если правом создания блока `stId` не обладает).

Реализация обобщенного функционала метода делегации доли владения состоит из двух составляющих:

```
data StakeConfiguration e mw rs proof stId = StakeConfiguration
   { scGetStake :: stId -> ERoComp e mw rs (Ratio Int)
   , scEligibleToForge :: stId -> proof -> ERoComp e mw rs ()
   }
```

Листинг 3.1 — Конфигурация обобщенного функционала PoS

– объект `BlockIntegrityVerifier`, проверящий что первая транзакция блока имеет специальный тип `blockStart`;
– валидатор для транзакции типа `blockStart`, проверяющий, что участник системы, чья подпись блока хранится в транзакции, имеет право выпуска блока при текущем состоянии системы.

Для реализации валидатора следует использовать метод `scEligibleToForge` конфигурации, который полностью энкапсулирует всю информацию, предоставляемую алгоритмом консенсуса. Следует особенно отметить, что с использованием конфигурации из листинга 3.1, реализации валидатора и объекта `BlockIntegrityVerifier` получатся обобщенными, т.е. подходящими для использования с любыми алгоритмами консенсуса, предоставляющими конфигурацию. Тип конфигурации же является достаточно абстрактной, чтобы его можно было удовлетворить, используя любой из алгоритмов, использующих метод доказательства доли владения.

3.2. Делегация доли владения

Делегация доли владения является важным механизмом в алгоритмах консенсуса, реализующих метод доказательства доли владения. Она предоставляет функционал передачи права выпуска блока от одного участника системы другому, позволяя:

– получателю прав выпускать блоки в тех случаях, когда владелец прав имеет право выпуска блока;
– владельцу прав производить манипуляции с его долей владения без каких-либо ограничений, в том числе повторно передавать право выпуска блока (третьему участнику).

Существует два подхода к реализации делегации, которые ведут к двум различающимся типам делегации: легковесной и тяжеловесной, которые будут рассмотрены в соответствующих подразделах.

Что существенно, оба эти подхода являются модификациями алгоритма консенсуса, использующего метод PoS, применимыми к любому алгоритму такого вида. Реализации обоих подходов модифицируют конфигурацию метода делегации доли владения и не предъявляют к системе каких-либо дополнительных требований. Это позволяет использовать реализацию обоих типов делегации в любой системе с алгоритмом, использующей метод PoS без внесения модификаций в реализацию механизма делегации.

3.2.1. Легковесная делегация

Легковесная делегация представляет из себя механизм делегации, не требующих хранения каких-либо дополнительных данных в состоянии системы.

В случае когда один участник сети (делегирующий) имеет намерение передать другому участнику (делегату) право выпуска блока, делегирующий участник создаёт специальный объект – делегационный сертификат, который хранит внутри себя следующую информацию:

- публичный ключ делегирующего;
- публичный ключ делегата;
- значение целочисленного счётчика;
- подпись пары из публичного ключа делегата и значения счётчика ключом делегирующего.

Делегирующий каким-либо образом передаёт сертификат делегату. Когда делегат выпускает блок, используя право выпуска блока делегирующего, делегат вкладывает в доказательство блока сертификат. Участники, проверяющие блок, должны проверить подпись сертификата делегирующим, проверить корректность значения счётчика.

На значение счётчика накладываются следующие ограничения:

- начальное значение счётчика для ключа делегирующего равно 0;
- каждый последующий сертификат, созданный делегирующим, должен содержать значение счётчика, большее известного на текущий момент системе;
- блок считается некорректным, если к нему приложен сертификат, значение счётчика которого меньше наибольшего использованного в ранее принятых блоках;

– правило сравнения двух цепей должно быть модифицировано таким образом, что если две цепи различаются только последним блоком, оба последних блока созданы с помощью сертификатов, выпущенных одним делегирующим участником, то предпочтение отдаётся блоку с большим значением счётика.

В листинге 3.2 представлена реализации легковесной делегации. Модифицируется конфигурация обобщенного функционала PoS (функция `lightDlgStakeConf`): создатель блока имеет право выпуска блока в том числе если к блоку приложен сертификат, в котором он выступает делегатом, а делегирующий имеет право выпуска блока. Также модифицируется правило выбора цепи (функция `lightDlgChainComparator`).

```
data LightDlgCert stId signature =
  LightDlgCert
    { ldcIssuer :: stId
    , ldcSignature :: signature
    , ldcCounter :: Int
    }

data LightDlgSTag
type LightDlgKV stId = '( stId, Int )

newtype LightDlgToSign stId = LightDlgToSign stId

data LightDlgError stId signature
   = LightDlgCorruptedCert (LightDlgCert stId signature)

lightDlgStakeConf
  :: ( Ord stId
     , Signed stId signature (LightDlgToSign stId)
     , HasPrism proof (LightDlgCert stId signature)
     , HasException e (LightDlgError stId signature)
     , HasCap (StateTag LightDlgSTag) rs
     )
  => Proxy (stId, signature)
  -> StakeConfiguration e mw rs proof stId
  -> StakeConfiguration e mw rs proof stId
lightDlgStakeConf = (..)

lightDlgChainComparator
  :: HasPrism header (LightDlgCert stId signature)
  => ChainComparator header
  -> ChainComparator header
lightDlgChainComparator = (..)
```

Листинг 3.2 — Функционал легковесной делегации

Легковесная делегация не позволяет участникам системы получить доступ к информации о произошедших в системе делегациях (факт делегации перед выпуском блока известен лишь делегирующему и делегату), поскольку

не использует состояние системы. Это значительно сужает спектр применения такого типа делегации. Однако легковесная делегация отлично подходит для оперативного менеджмента секретных ключей, в частности для реализации схемы холодного хранилища.

Большинство блокчейн-систем накладывает на участников сети поддерживать узлы сети, выпускающие блоки работающими значительную часть времени. Адреса этих узлов известны другим участникам сети и в случае проведения успешного взлома, секретные ключи, использующиеся для выпуска блоков узлом сети, могут попасть в руки злоумышленника.

Легковесную делегацию можно использовать для решения проблемы следующим образом: участник сети создаёт два секретных ключа: “горячий” ключ `hotKey` и “холодный” ключ `coldKey`. С помощью холодного секретного ключа генерируется сертификат делегации (для делегации права выпуска блока с `hotKey` на `coldKey`), который вместе с горячим секретным ключом устанавливается на узел сети, выпускающий блоки от лица участника. Холодный ключ кладется в некоторое хранилище, предпочтительно без доступа к сети. Публичная компонента холодного ключа используется для идентификации участника сети. В случае если узел, выпускающий блоки, будет взломан, участник сети сможет с использованием холодного ключа сделегировать право выпуска блока на новый ключ `hotKey2`, запустить новый узел, выпускающий блоки, тем самым полностью нейтрализовав действия злоумышленника.

3.2.2. Тяжеловесная делегация

Тяжеловесная делегация представляет из себя еще один механизм делегаций, значительно отличающийся от легковесной делегации. Тяжеловесная делегация хранит информацию о совершенных делегациях в состоянии системы, что непосредственно приводит к ряду следствий:

– информация о совершенных делегациях доступна всем участникам сети;
– передача права выпуска блока требует проведения транзакции к состоянию;
– есть возможность реализации многоуровневой делегации.

Делегационный сертификат тяжеловесной делегации включает в себя те же данные, что и сертификат легковесной:

– публичный ключ делегирующего;
– публичный ключ делегата либо пометка об отсутствии делегата;
– значение целочисленного счётчика;
– подпись пары из публичного ключа делегата и значения счётчика ключом делегирующего.

Следует заметить, что у делегационного сертификата тяжеловесной делегации может не быть делегата (вместо значения делегата может стоять соответствующая пометка). В этом случае сертификат отменяет передачу права выпуска блока, возвращая его владельцу.

При рассмотрении семантики применения блоков тяжеловесная делегация имеет ряд особенностей по сравнению с легковесной:
– после совершения делегации, делегирующий лишается права выпуска блоков (оно полностью передано делегату);
– сертификат нет нужды прикладывать к созданному блоку, т.к. информация из него уже содержится в состоянии;
– нет нужды модифицировать правило выбора цепи;

Механизм тяжеловесной делегации целесообразно обобщить до многоуровневой делегации: делегату B (которому делегировал право выпуска блока участник A) позволяется выступать делегирующим, создавая новый сертификат делегации и передавая право выпуска блока третьему участнику C. В конфигурации делегации тогда задаётся целочисленный параметр `maxDlgHeight`, ограничивающий максимальное число переделегаций.

Опр. 3.1. Граф делегации – ациклический направленный (невзвешенный) граф. Вершины графа – публичные ключи участников. Каждый сертификат в системе порождает ровно одно ребро в графе, ребро направлено от делегирующего к делегату (из каждой вершины исходит не более одного ребра).

В листингах 3.3, 3.4 представлена реализация тяжеловесной делегации. Модифицируется конфигурация обобщенного функционала PoS (функция `heavyDlgStakeConf`): создатель блока имеет право выпуска блока в том числе если в состоянии есть информация о делегации, в которой он выступает делегатом, а делегирующий имеет право выпуска блока. Кроме того, метод конфигурации `scGetStake` изменяется таким образом, чтобы учитывать информацию о тяжеловесных делегациях (в случае делегации от участника A участнику B, значение доли владения участника B должно быть увеличино на значение

доли владения участника A, а доля владения участника A должна равняться нулю).

Тяжеловесная делегация определяет новый тип транзакции и соответствующий ему валидатор.

Тяжеловесная делегация задаёт две компоненты состояния `delegate :: Map stId stId` и `issuers :: Map stId (Set stId)`. Компонента состояния `delegate` описывает дуги графа делегации, компонента `issuers` по ключу u хранит множество всех вершин v, для которых существует путь из u в v в графе делегации). Данные из компоненты `delegate` используются для поддержки графа делегации (приема и валидации транзакций), компонента `issuers` используется для поддержания алгоритма консенсуса, в частности в модификации конфигурации метода делегации доли владения.

```
data HeavyDlgCert stId signature =
   HeavyDlgCert
     { hdcIssuer :: stId
     , hdcDelegate :: Maybe stId
     , hdcCounter :: Int
     }

data HeavyDlgDelegateSTag
type HeavyDlgDelegateKV stId = '( stId, stId )

data HeavyDlgIssuersSTag
type HeavyDlgIssuersKV stId = '( stId, Set stId )

newtype HeavyDlgToSign stId = HeavyDlgToSign (Maybe stId)

data HeavyDlgTxId = HeavyDlgTxId

newtype HeavyDlgCertProof signature = HeavyDlgCertProof signature

data HeavyDlgValidationError
   = HvDlgDelegationCycle
   | HvDlgDelegationTooDeep
   | HvDlgWrongOldIssuers
   | HvDlgWrongOldDelegate
   | HvDlgWrongDelegates
   | HvDlgWrongIssuers
```

Листинг 3.3 — Типы для функционала тяжеловесной делегации

3.2.3. Комбинация двух видов делегаций

Для реализации блокчейн-системы с функциональностью делегации наиболее целесообразно использовать комбинацию двух типов делегации. Легковесная делегация удобна для оперативного менеджмента ключей: замена

```
heavyDelegationStakeConf
  :: ( Ord stId
     , HasException e StatePException
     , HasCap (StateTag HeavyDlgIssuersSTag) rs
     )
  => Proxy (stId, signature)
  -> StakeConfiguration e mw rs proof stId
  -> StakeConfiguration e mw rs proof stId
heavyDelegationStakeConf = (..)

heavyDlgValidator
  :: ( HasPrism proof (HeavyDlgCertProof signature)
     , HasExceptions e
       [ HeavyDlgValidationError, StatePException, TxValidationError]
     , Ord stId
     , IdStorage txIds HeavyDlgTxId
     , Signed stId signature (HeavyDlgToSign stId)
     )
  => Proxy (stId, signature, txIds)
  -> Int
  -> Validator e mw '[ StateTag HeavyDlgIssuersSTag
                     , StateTag HeavyDlgDelegateSTag ] proof
heavyDlgValidator p maxDlgHeight = (..)
```

Листинг 3.4 — Функционал тяжеловесной делегации

одного ключа другим производится мгновенно, не накладывая дополнительных требований на другие части системы.

Тяжеловесная делегация предоставляет функционал, который отсуствует в легковесной (многоуровневая делегация, согласованность, доступ к информации о делегации для всех участников сети), однако не позволяя решение задачи оперативного менеджмента ключей (смена делегации требует принятия в блок специальной транзакции, достижения консенсуса об этом блоке участниками сети).

Предлагаемая схема проиллюстрирована на рисунке 3.1 (легковесная делегация обозначена пунктирной стрелкой): на узлах, выпускающие блоки, сохраняется ключ `hotKey`. Выпускается соответствующий легковесный сертификат, делегирующий право выпуска блока с ключа `coldKey` на ключ `hotKey`. Ключ `coldKey` может храниться в некотором закрытом хранилище и использоваться только при необходимости сменить `hotKey`. В то же время `coldKey` может выступать ключом, на который проихводится делегация тяжеловесного типа с других ключей (кол-во уровней делегации ограничено только параметром `maxDlgHeight`).

Таким образом, использование легковесной делегации позволяет реализовать оперативный менеджмент ключей, тогда как использование тяжело-

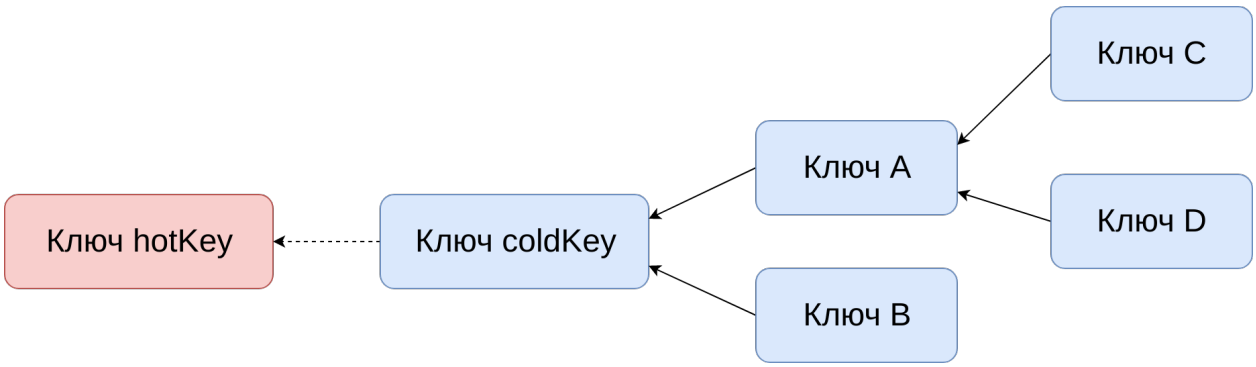

Рис. 3.1 — Комбинация легковесной и тяжеловесной делегаций

весной делегации позволяет системе прямо или косвенно пользоваться информацией о совершенных делегациях (что неявно используется в построении системы обновлений, равно и построении некоторых алгоритмов консенсуса).

3.3. Система обновления

Функционал обновления системы объединяет в себе набор механизмов, использующихся системой для:

– извещения пользователей о доступных обновлениях ПО;
– обновления протоколов системы, в т.ч. алгоритма консенсуса и его параметров;

Во всякой современной блокчейн-системе предусмотрена поддержка обновления ПО и протокола. Во многих блокчейн-системах поддержка обновления системы сводится к версионированию блоков (т.о. при обновлении версии блока клиент знает о необходимости обновления локальной версии ПО), в других же системах, например в Bitcoin [1], существует ряд встроенных механизмов, позволяющих проводить ряд модификаций протокола с поддержкой совместимости с ранними версиями ПО. Ни одна из существующих систем не реализует систему согласования обновления протокола или ПО между участниками сети.

Опр. 3.2. Софт-форк – обновление протокола, поддерживающее совместимость с существующими версиями ПО.

Опр. 3.3. Хард-форк – обновление протокола, требующее обновления ПО большинством участников сети для дальнейшего функционирования сети.

Под обновлением ПО мы понимаем обновление системы, не требующее обновление протокола (т.е. обновление только ПО, используемого узлами сети).

Поддержка хард-форков является интереснейшей и важной задачей: реализация посредством софт-форков возможно только для ограниченного спектра обновлений протокола, до той степени, до которой создатели оригинального протокола предусмотрели эту возможность. В широко использующейся системе необходимость проведения хард-форка рано или поздно себя проявит, как следствие поддержка их важна. В настоящей работе поддержка хард-форков не будет рассмотрена подробно, однако её можно реализовать, используя идеи, схожие с теми что будут сформулированы в настощем разделе по отношению к софт-форкам, с применением механизма боковых цепей (англ. side chains [33]). Основной сложностью в моделировании и проведении хард-форка притом является недопущение возможного разделения сети на две и более несвязных сегментов.

3.3.1. Дизайн системы с поддержкой софт-форков

Возможность выражения обновления протокола как софт-форка опирается на закладывание в построенную систему возможности расширения её функциональности без изменения используемых протоколов (это касается протоколов взаимодействия по сети, сериализации, реализации алгоритма консенсуса и т.д.). Основными двумя механизмами, которые можно использовать в дизайне протоколов системы, которые позволяют оставить пространство для расширения функционала в будущем, являются:

– параметризация протоколов;
– использование версионируемых структур данных.

Под параметризацией протоколов понимается вынесение важных параметров работы системы в некоторую объединенную конфигурацию, которую затем можно без особых усилий обновлять. Многие параметры, от которых зависит протокол сети, можно вынести в такую конфигурацию: размер блока (дебаты из-за увеличения размера блока в сети Bitcoin не утихали многие месяцы, что в итоге привело к разделению сети на два сегмента), формула вычисления комиссии за транзакцию, временная задержка выпуска блоков и

т.д. Вынесение их в конфигурацию на этапе дизайна системы значительно облегчает задачу обновления системы в будущем.

Конфигурация с параметрами системы может быть сделана частью объекта обновления системы, и в случае если обновление будет принято большинством участников сети, все участники сети смогут использовать обновленную версию протокола без обновления ПО (в случае, если новая версия протокола содержит исключительно обновление конфигурации параметров).

Использование версионируемых структур данных само по себе немногим помогает в реализации софт-форков. Однако идею версионирования данных можно развить соответствующим образом. Предположим есть клиент c, использующий устаревшую версию протокола, в котором поддерживались типы адреса $A1$, $A2$, но отсутствовала поддержка типа адреса $A3$. Предположим клиенту c приходит блок новой версии, содержащий транзакции, которые работают с адресом $A3$.

Очевидно, полную валидацию транзакций (особенно в случае списания средств с адреса типа $A3$) клиент c выполнить не может. Клиент c по-прежнему может провалидировать остальные транзакции в блоке, также он может соответствующим образом изменить записи о балансах в предположении что транзакция типа $A3$ верна. Предположение о верности $A3$ может быть ложным, однако если в дальнейшем большинством участников системы транзакция будет подтверждена и если системой гарантируется что большинство участников системы используют последнюю версию протокола, это предположение является корректным. В противном случае, если блок с транзакцией типа $A3$ будет в дальнейшем отклонен большинством участников сети, клиент c может откатить блок с невалидной транзакцией.

3.3.2. Механизм проведения софт-форков

Опишем механизм обновления, сформулированный в системе, основанной на алгоритме консенсуса, использующего метод доказательства доли владения.

Опр. 3.4. Объект обновления – объект данных, однозначно описывающий обновление системы.

Объект обновления системы должен включать в себя:

– обновленную конфигурацию параметров системы;
– хэши бинарных объектов новой версии ПО;
– обновленный список версий версионируемых структур данных;
– версию протокола;

Версию протокола предлагается обозначать тройкой чисел $(Maj,\ Min,\ Alt)$. Значение Maj обновлется только в случае обновления протокола типа хард-форк. Значение Min обновляется при обновлении протокола типа софт-форк. Значение Alt служит для различения конкурирующих обновлений системы.

Объект обновления публикуется для участников системы, после чего происходит голосование. Голоса участников взвешиваются в соответствии с их долей владения (для этого следует использовать метод `scGetStake` конфигурации обобщенного функционала метода доказательства доли владения). Каждый участник может проголосовать “за” или “против” обновления.

В случае, если в какой-то момент $> 50\%$ участников проголосуют “за” или “против” обновления, обновление перейдет в статус подтвержденного, либо отклоненного.

Возможные изменения статуса обновления показаны в диаграме 3.2.

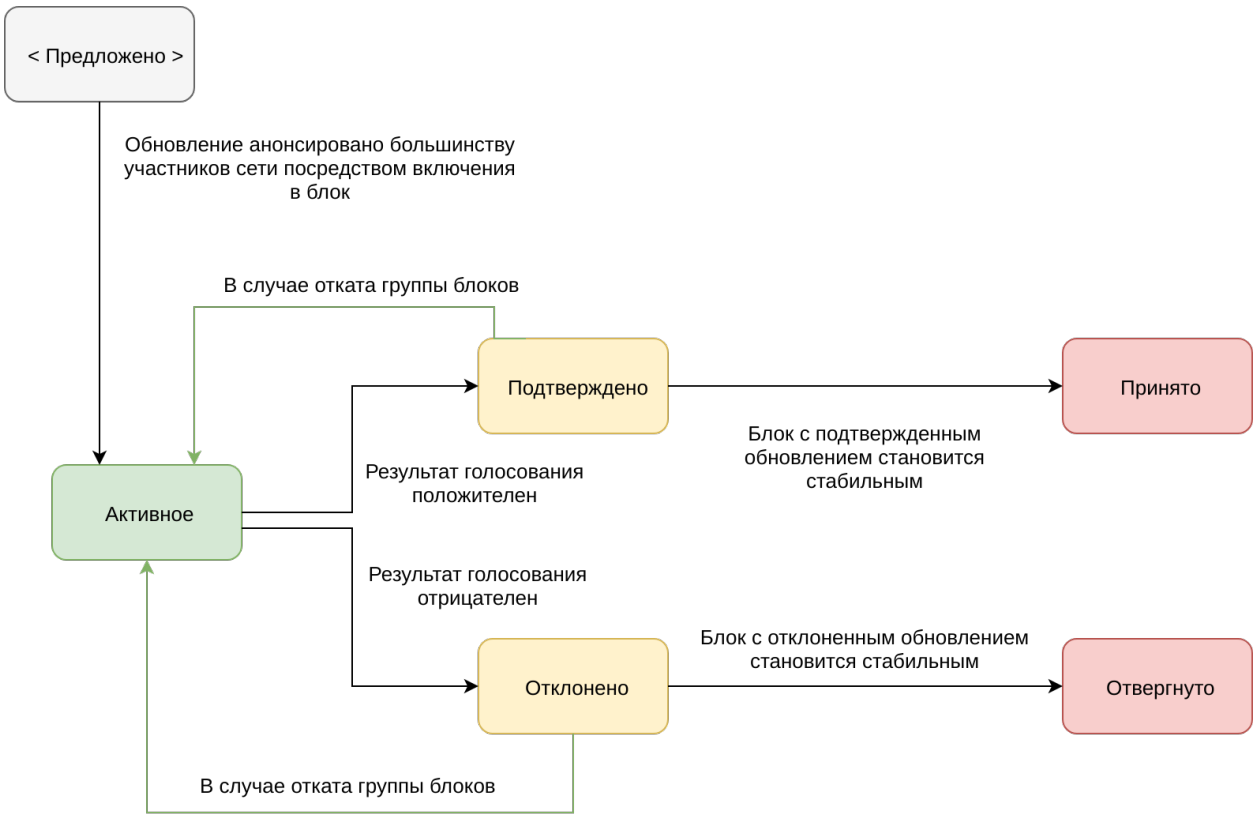

Рис. 3.2 — Изменения статуса обновления протокола

После подтверждения обновления большинством участников сети и стабилизации блока с последним имеющим значение голосом, версия протокола принимает статус соревнующейся. Она станет примененной только в тот момент, когда подавляющее число владельцев доли (параметр задается произвольно, но на практике следует использовать параметр не менее 80%) обновит версию ПО и выпустит блок, подтверждающий факт обновления ПО до версии, поддерживающей новую версию протокола.

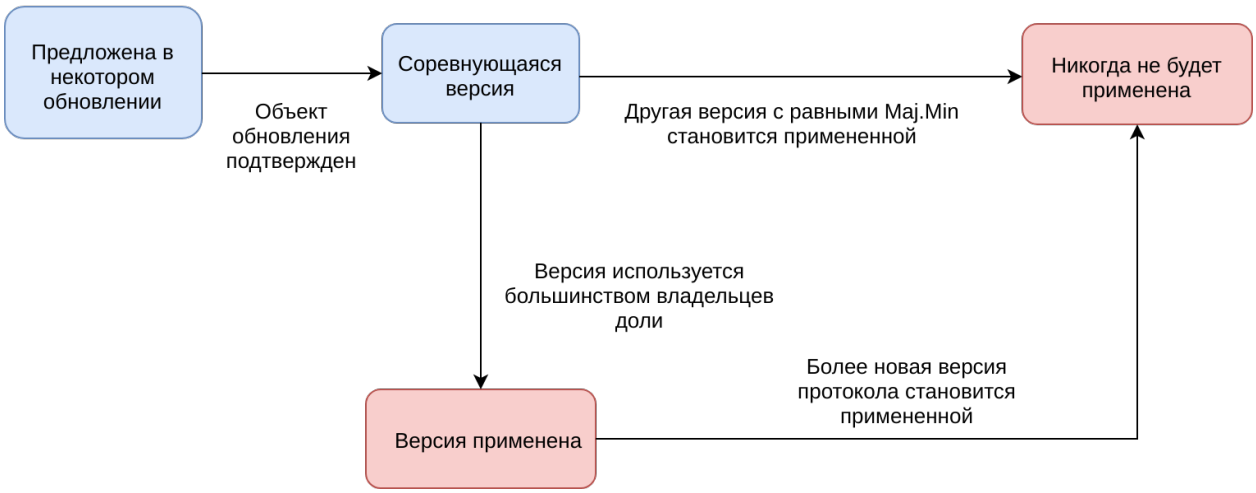

Рис. 3.3 — Изменения статуса версии протокола

3.3.3. Реализация механизма проведения софт-форков

Для эффективной реализации механизма проведения софт-форков требуется определить период времени, по прошествии которого будет производиться итерация по всем голосам, соответствующим каждому из активных обновлений, подсчитывать суммарную долю владения всех участников, проголосовавших “за” или “против” и в случае если количество голосов достаточное, перевести обновление в статус подтвержденного, а версии протокола – в статус конкурирующей.

Описанные действия не представляет труда описать в рамках предлагаемой модели с использованием конфигурации обобщенного функционала метода делегации доли владения с использованием модели состояния и модели обработки блоков.

Резюме

ГЛАВА 4. **МОДЕЛЬ КРИПТОВЛЮТЫ**

Под термином криптовалюта принято понимать блокейн-систему, основной сферой применения которой является проведение финансовых операций.

Базовой функциональностью криптовалют является поддержка балансов пользователей или аккаунтинг. Рассмотрению реализации аккаунтинга посвящен раздел 4.1. Кроме того, большинство современных криптовалют предлагают широкий инструментарий для реализации финансовых контрактов. Реализации контрактов в предлагаемой модели криптовалюты посвящены разделы 4.2, 4.3.

4.1. Аккаунтинг

Существует две модели аккаунтинга, получившие наибольшее распространение в криптовалютах: аккаунтинг, поддерживающий UTxO записи и аккаунтинг, поддерживающий счета пользователей. Обе модели формализованы в терминах теории типов в работе Chimeric Ledgers [34], в настоящем разделе будут предложены практически идентичные формализации обоих моделей аккантинга на основе модели блокчейн-системы.

Для реализации криптовалюты можно использовать любую из них (возможна трансляция из одной модели в другую [34]), описанные в настоящем разделе модели можно применять независимо друг от друга, причем другие функциональности блокчейн-системы в предлагаемых моделях блокчейн-системы и криптовалюты не зависят от выбора в пользу одной из моделей аккаунтинга.

4.1.1. Аккаунтинг, поддерживающий UTxO записи

Всякая транзакция в моделе аккаунтинга, поддерживающего UTxO записи, представляет из себя два списка: список входов типа `TxIn` и список выходов типа `TxOut`. Вход представляет собой пару из идентификатора одной из ранее произошедших транзакций и индекса выхода в ней. Выход представляет собой пару из адреса и количества монет, которые должны быть на этот адрес перечислены.

```haskell
data TxIn txId = TxIn
  { tiId :: txId
  , tiIndex :: Int
  }

data TxOut addr coins = TxOut
  { toAddr :: addr
  , toCoins :: coins
  }
```

Листинг 4.1 — Типы UTxO аккаунтинга

Типы входа и выхода представлены в листинге 4.1 (в качестве типа `coins` можно поставить любой целочисленный тип), конструкция транзакции проиллюстрирована на рисунке 4.1.

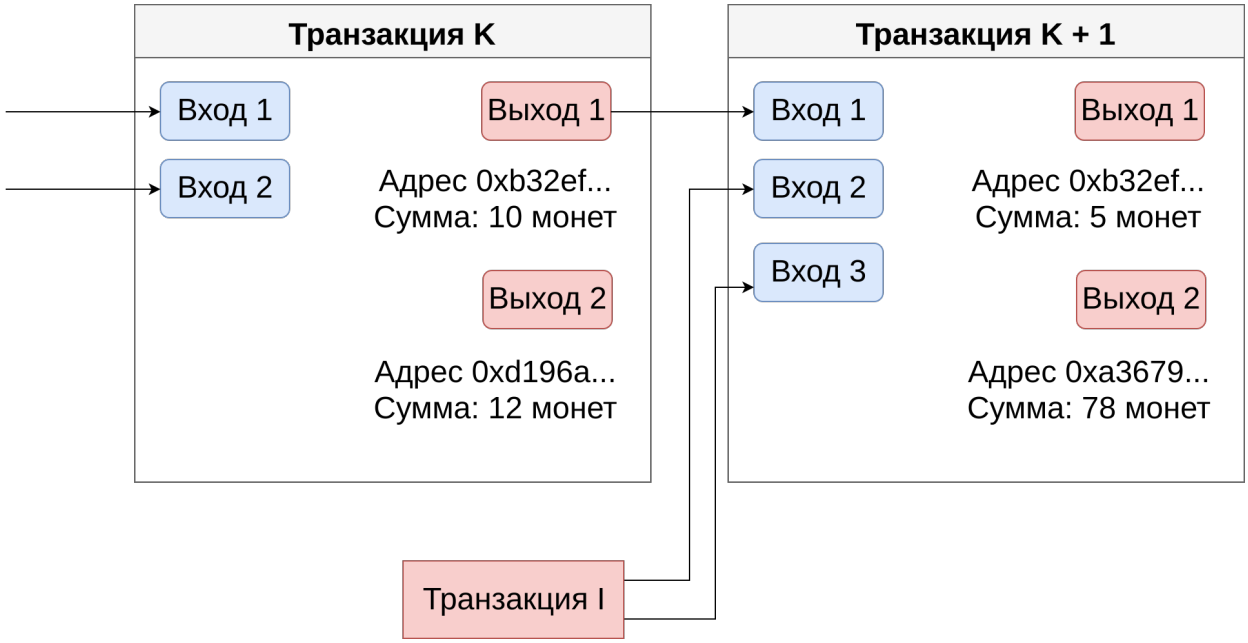

Рис. 4.1 — Транзакции в UTxO аккаунтинге

В моделе аккаунтинга, поддерживающий балансы пользователей посредством записей UTxO, состояние системы хранит отображение из объекта входа `TxIn` в объект выхода `TxOut` для всех выходов, которые еще не были использованы как входы какой-либо из транзакций. Подсчёт баланса пользователя, т.е. суммы средств, принадлежащих адресу пользователя, включает итерацию по всем записям в отображении и фильтрацию записей, поле адреса в которых равно искомому. Модель аккаунтинга названа в честь аббревиатуры UTxO, переводимой как “непотраченный выход транзакции” (англ. Unspent Transaction Output).

В листингах 4.2, 4.3 представлен фрагмент реализации UTxO аккаунтинга, использующая модель блокчейн-системы. Интересно, что валидатор

```
data UtxoTx = UtxoTxId

data UtxoTxSTag
type UtxoTxKV txId addr coins = '( TxIn txId, TxOut addr coins )

data UtxoTxValidationException id txId
   = TxInsAndTxOutsIntersect
   | NoWitness (TxIn txId)
   | InvalidWitness (TxIn txId)
   | InputsSumLessThanOutputsSum

data UtxoTxConfiguration mw rs proof witness = UtxoTxConfiguration
   { verifyWitness :: StateTx mw rs proof -> witness -> Bool
   }

type UtxoTxExtraction txId addr coins witness e mw rs =
   ( HasPrism proof (TxIn txId ~> witness)
   , HasException e StatePException
   , Ord (TxIn txId)
   , HasCap (StateTag UtxoTxSTag) rs
   )
```

Листинг 4.2 — Реализация UTxO аккаунтинга

UTxO транзакции разбивается на несколько предвалидаторов, которые объединяются, используя определения моноидальной композиции для типа.

4.1.2. Аккаунтинг, поддерживающий счета пользователей

Аккаунтинг, поддерживающий счета пользователей, хранит значения балансов пользователей непосредственно (а не опосредованно, как это делается в модели аккаунтинга UTxO). Транзакция представляет собой изменение в балансах аккаунтов. Помимо баланса, каждый аккаунт хранит целочисленное значение $nonce$, являющееся счётчиком операций, совершенных с аккаунтом. Для подтверждения операции над аккаунтом, транзакция, стремящаяся потратить средства с аккаунта, должна использовать большее значение $nonce$, чем использованное ранее для данного аккаунта.

В листинге 4.4 представлены типы, использующиеся аккаунтингом, поддерживающего счета пользователей. В состоянии хранится отображение из адреса (тип `AccountId addr`) в тип `Account coins`. Более подробное рассмотрение реализации будет опущено, т.к. практически полностью повторяет реализацию UTxO аккаунтинга.

```
utxoTxValidator
   :: ( UtxoTxExtraction txId addr coins witness e mw rs proof
     , Num coins, Ord coins, Ord txId
     , IdStorage txIds UtxoTx
     , HasExceptions e
        [ UtxoTxValidationException id txId
        , StructuralValidationException id
        ]
     )
   => Proxy txIds
   -> UtxoTxConfiguration mw rs proof witness
   -> Validator e mw rs proof
utxoTxValidator p utxoConfig =
   mkValidator txType
      [ utxoTxStructuralPreValidator @txId @addr @coins @witness
      , witnessPreValidator @txId @addr @coins utxoConfig
      , sumInputsNotLessOutputsPreValidator @txId @addr @coins @witness
      ]
 where
   txType = StateTxType ( getId (Proxy @txIds) UtxoTxId )
```

Листинг 4.3 — Валидатор UTxO транзакции

```
data AccountTx = AccountTxId

newtype AccountId addr = AccountId addr

data Account coins = Account
   { aBalance :: coins
   , aNonce :: Int
   }

data AccountTxSTag
type AccountTxKV addr coins = '( AccountId addr, Account coins )
```

Листинг 4.4 — Типы аккаунтинга, поддерживающего счета пользователей

4.1.3. Взаимосвязь аккаунтинга и метода доказательства доли владения

В методе доказательства доли владения используется понятие доли владения – пропорции в которой тот или иной участник системы владеет системой. Стандартным подходом в реализации систем, построенных с использованием метода PoS является ассоциация долей владения с балансами в аккаунтинге. Т.е. $stake\ (u) \equiv \frac{balance\ (u)}{total\ balance}$, что является логичным развитием идеи доли владения.

Однако в реализации такой ассоциации легко допустить ряд ошибок. В предыдущих разделах не рассматривалось представление адресов, транзакций в виде сериализованных значений (т.к. рассмотрение подобных подробностей в целом выходит за рамки настоящей работы), в настоящем подраз-

деле мы рассмотрим представление адреса в транзакции и введем соответствующее представление адреса в системе с аккаунтингом, основанной на алгоритме консенсуса, реализующем метод PoS.

Всякая транзакция хранит в себе объект типа `proof` и в случае аккаунтинга этот объект содержит доказательство корректности списания средств с адреса автором транзакции. Это доказательство в простейшем случае представляет собой цифровую подпись, сделанную ключом автора транзакции. Причем адрес, на котором лежат средства в таком случае представляет собой значение применения некоторой криптографически-стойкой хэш-функции к ключу автора: $Addr \equiv Hash\ PublicKey$, $validate\ (pk,\ Proof\{signature\}) = verify\ (pk,\ signature)$. Публичный ключ “оборачивается” в хэш-функцию для получения адреса из соображений безопасности: в случае взлома схемы электронной подписи, списать средства с адреса будет по-прежнему практически невозможно, по крайней мере пока также не будет найдена уязвимость в строении хэш-функции.

Если ассоциировать балансы и доли владения наивным образом, т.е. подписывать блоки ровно тем же ключом, на соответствующих которому адресах хранятся средства, эта дополнительная защита в виде хэширования теряет свою функцию: при создании блока публичный ключ адреса, на котором хранятся средства, будет опубликован. От данного упущения в частности страдают криптовалюты EOS [27], NEM [22] (спецификации которых рассмотрены в разделе 1.3).

Единственным способом поддерживать те же гарантии безопасности в криптовалюте с аккаунтингом, основанной на PoS методе, является разделение секретных ключей. Один ключ, ассоциированный с балансом (“балансовый” ключ) позволяет владельцу манипулировать его счётом, посылая денежные транзакции. Другой ключ, ассоциированный с долей владения (“стейковый” ключ) позволяет владельцу манипулировать его долей владения (например, совершать делегационные транзакции), выпускать блоки, используя его долю владения.

Однако, адрес по-прежнему должен ссылаться как на балансовый ключ, так и на стейковый ключ. Причем публичная компонента стейкового ключа может быть раскрыта, тогда как публичная компонента балансового ключа

раскрыта быть не должна. В листинге 4.5, а также на рисунке 4.2 представлено строение адреса, отвечающего поставленной задаче.

```
type Hash x = (...)

newtype StakePk = StakePk (...)
newtype BalancePk = BalancePk (...)


type Address = (StakePk, Hash (BalancePk, StakePk))
```

Листинг 4.5 — Адрес, ссылающийся и на стейковый, и на балансовый ключи

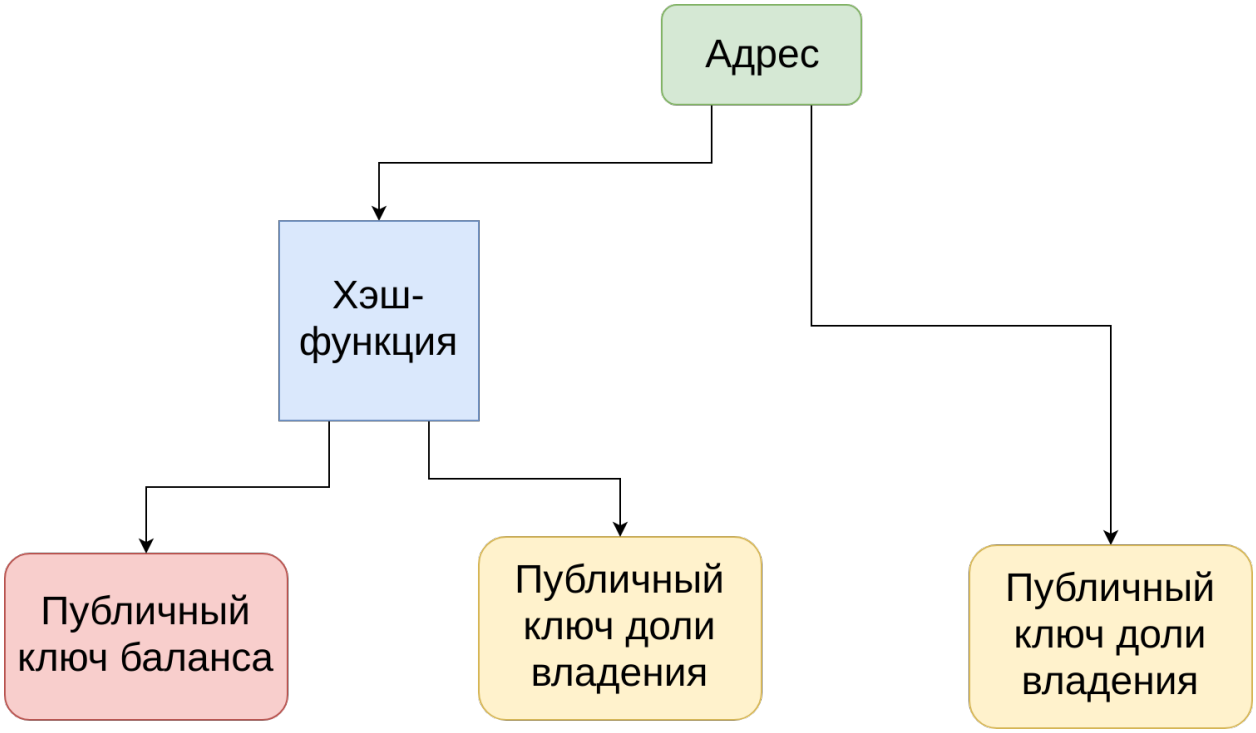

Рис. 4.2 — Адрес, ссылающийся на стейковый, и на балансовый ключи

Предлагаемая схема репрезентации адреса реализует поставленную задачу: стейковый ключ общедоступен, балансовый ключ – нет. В данной схеме балансовый ключ будет раскрыт только в момент процессинга транзакции, переводящей средства на другой адрес. Также следует заметить, что невозможно использовать в качестве стейкового публичного ключа, указанного в адресе, стейковый ключ, не соответствующий хэшу. Такой адрес будет принят системой, однако расходовать средства с него не будет возможно (возможность подстановки произвольного стейкового ключа в адрес была сделала бы возможным проведение ряда атак, изложение которых выходит за рамки настоящей работы).

4.2. Поддержка умных контрактов

Опр. 4.1. Умный контракт – программа, определяемая в контексте некоторой блокчейн-системы, запускаемая в окружении, в котором ей доступны операции чтения и записи данных в состояние блокчейн-системы.

Умные контракты были впервые предложены криптовалютой Ethereum [4] и на сегодняшний день является одной из наиболее распространенных функциональностей, реализуемых в новых блокчейн-системах. Они используются для реализации финансовых контрактов (многоступенчатых процедур, представляющих нетривиальную финансовую сделку между двумя и более участниками), равно и для реализации систем распределенных вычислений.

Блокчейн-система предоставляет некоторую виртуальную машину. Примитивы виртуальной машины предоставляют как стандартные операции манипуляции с данным, контроля потока исполнения, так и специальные операции, позволяющие чтение произвольных данных, хранящихся в локальном состоянии узла системы, запись в состояние.

Контракты сохраняются в состоянии системы специальным типом транзакции, с контрактом ассоциируется область в адресном пространстве системы. Всякий раз когда в систему поступает транзакция, использующая адрес контракта, происходит выполнение контракта, в результате которого происходит подтверждение либо отклонение транзакции, а также возможно изменение некоторых других частей состояния.

Для реализации умных контрактов на основе предложенной модели блокчейн-системы следует использовать механизм дополнения транзакции. Контракт представляет собой функцию дополнения, которая по предложенному описанию транзакции генерирует список изменений в состояние, которые должны произойти. Следует заметить, что умный контракт общего вида (как в частности контракты, использующиеся в системе Ethereum [4]) требуется исполнить для транзакции tx_1, чтобы перейти к валидации следующей транзакции tx_2, как следствие параллелизация обработки множества транзакций в общем случае не представляется возможным. Однако, дополнение транзакции общего вида полностью удовлетворяет задаче поддержки умных контрактов.

Следует также отметить, что прежде чем контракт будет запущен для входной транзакции и текущего состояния сети, невозможно знать, какие изменения в состоянии применение контракта породит (таким образом, изменения состояния, производимые контрактов указываются неявно).

4.3. Контракты с локальным состоянием

Опр. 4.2. Контракт с локальным состоянием – умный контракт, область записи которого ограничена некоторой областью общего состояния системы, в которую недоступны чтение и запись для любых других контрактов (и прочих подсистем блокчейн-системы).

Название контрактов с локальным состоянием отражает их суть: контракт поддерживает некоторое лишь ему доступное состояние, использует его для валидации транзакций. Основное различие с умными контрактами состоит в том, что контракт с локальным состоянием не имеет возможности произвольно изменять общее состояние системе, как следствие все изменения общего состояния должны быть частью объекта изменения транзакции, обрабатываемой контрактом.

Ограничение выразительной мощности умных контрактов достаточно существенно: умный контракт теперь не может, например, инициировать новую денежную транзакцию, его роль сводится исключительно к валидации текущей транзакции (для чего он может произвольно использовать собственное состояние). Однако для реализации абсолютного большинства практических применений умных контрактов (в первую очередь для реализации финансовых контрактов) такое ограничение не является существенным. Кроме того, строение контракта с локальным состоянием гарантирует то, что все изменения балансов (хранение балансов не относится к локальному состоянию какого-либо из контрактов) будут явно указаны в транзакции, что в реализации финансовых контрактов является плюсом (явно указываются финансовые транзакции, которые будут совершены).

Введение контрактов с локальным состоянием мотивировано в первую очередь желанием применить дополнение транзакции ограниченного вида к умным контрактам. Как было рассмотрено в разделе 2.3, дополнение тран-

закции ограниченного вида предлагает ряд интересных свойств для эффективной обработки множества транзакций, ассоциированных с умными контрактами. В частности использование дополнения транзакции ограниченного вида позволяет обрабатывать множество N транзакций в $O(N)$ потоков, каждый из которых сделает $O(T)$ действий, где T – максимальное количество контрактов, которые потребуется выполнить для дополнения произвольной транзакции. Следует заметить, что в случае применения ограничения, что многие системы можно моделировать таким образом чтобы параметр T был равен единице (это в частности верно для системы Ethereum).

Рассмотрим применение дополнение транзакции ограниченного вида к контракту с локальным состоянием. Дополнение транзакции ограниченного вида требует выполнения условия 2.3 для множества контрактов (как было отмечено в разделе 4.2, каждый контракт представляет собой функцию дополнение транзакции). Однако для контрактов с локальным состоянием можно сформулировать более сильное условие 4.1 (для множества всех контрактов в системе cs), которое на самом деле позволяет выполнять дополнение N транзакций в $O(N \cdot T)$ потоков, каждый из которых сделает $O(1)$ операций.

$$\forall c_i, c_j \in cs \quad in\ (c_i) \cap out\ (c_j) = \emptyset \tag{4.1}$$

Следует однако уточнить несколько деталей реализации, касающихся выполнения условия 4.1. Условие $in\ (c_i) \cap out\ (c_i) = \emptyset$ выполняется только в том случае, если контракт не читает из компоненты состояния, в которую производит запись (тогда как именно это контрактам с локальным состоянием и разрешается).

Реализация должны быть устроена таким образом, чтобы операции чтения в дополнении транзакции не производилось. Объект транзакции перед дополнением `rawTx` вместе с значениями, полученными из состояния, являются единственными параметрами, от которых зависит изменение локального состояния. Они доступны функции дополнения контракта и, соответственно, контракт может создать объект изменения состояния для локального сосотояния, воспользовавшись конструктором `Upd (v -> v)` объекта изменения состояния (см. листинг 2.6) для всех требующих изменения ключей. Непосред-

Таблица 4.1 — Сравнение контрактов с локальным состоянием с другими молями контрактов

	Скриптовые адреса	**Умные контракты**	**Контракты с л.с.**
Явное описание финансовых операций	+	-	+
Поддержка локального состояния	-	+	+
Возможность параллельного исполнения	+	-	+

ственный запуск контракта с вынесением вердикта о валидности транзакции должно произойти несколько позже, в функции-валидаторе.

Также интересно заметить, что транзакции, использующие скриптовые адреса, представленные в системе Bitcoin [10], являются частным случаем контрактов с локальным состоянием, в котором локальное состояние контракта по определению пусто.

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.
33. Enabling blockchain innovations with pegged sidechains / Adam Back, Matt Corallo, Luke Dashjr [и др.] // URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains. 2014.
34. Zahnentferner Joachim. Chimeric Ledgers: Translating and Unifying UTXO-based and Account-based Cryptocurrencies. Cryptology ePrint Archive, Report 2018/262. 2018. https://eprint.iacr.org/2018/262.