ПОСЛЕДОВАТЕЛЬНЫЕ МАШИНЫ АБСТРАКТНЫХ СОСТОЯНИЙ ОХВАТЫВАЮТ ПОСЛЕДОВАТЕЛЬНЫЕ АЛГОРИТМЫ*

Юрий Гуревич

Мы рассматриваем последовательные алгоритмы и формулируем набор постулатов — Последовательное время, Абстрактное состояние и Локальность решения[1]. Анализ постулатов приводит к понятию последовательной машины абстрактных состояний и теореме, сформулированной в заголовке статьи. Сначала исследуются последовательные алгоритмы, являющиеся детерминированными и неинтерактивными. Затем рассматриваются последовательные алгоритмы, которые могут быть недетерминированными и взаимодействовать со своим окружением.

ПРЕДИСЛОВИЕ К РУССКОМУ ПЕРЕВОДУ

В начале было любопытство. Что представляет собой эта новая (для меня) наука? Была первая половина 80-х годов прошлого столетия, когда я переходил из математики в информатику. Я решил, что предметом информатики являются в первую очередь алгоритмы. При этом термин "алгоритм" понимается в широком смысле. Операционные системы, языки программирования, компиляторы и т.д. являются специальными алгоритмами. Это привело меня к фундаментальной проблеме — что же в точности является алгоритмом? В данной статье, после краткого описания истории вопроса, предлагается решение фундаментальеой проблемы для случая последовательных алгоритмов. Оказалось, что, с точки зрения поведения, последовательные алгоритмы — это в точности после-

*Перевод статьи [31]. В списке литературы звездочкой обозначены работы, добавленные при переводе.

[1]Авторское название этого постулата Bounded-Exploration Postulate. Фактически он требует, чтобы алгоритм в процессе своей работы принимал решения, анализируя только часть текущего состояния вычислений. Учитывая аналогию с локальными алгоритмами минимизации булевских функций, было решено использовать термин "Локальность [принятия] решения" в этом переводе. — *Прим. перев.*

довательные машины абстрактных состояний, введённые автором в 80-х годах.

Данная статья явилась началом плодотворной серии работ. В статье "Abstract State Machines Capture Parallel Algorithms", ACM Transactions on Computational Logic 4:4 (2003), pp. 578–651, Андреас Бласс и автор представили решение фундаментальной проблемы для случая параллельных алгоритмов. И снова оказалось, что, с точки зрения поведения, параллельные алгоритмы это в точности параллельные машины абстрактных состояний, введенные автором в 1980-х годах. Несколько дополнительных статей этой серии находится в разной степени готовности.

Решение данной фундаментальной проблемы нашло практическое приложение. Машины абстрактных состояний могут быть использованы для спецификации программного обеспечения. Это привлекло внимание компании Microsoft, где автор создал исследовательскую группу, работающую — в сотрудничестве с рабочими группами в разных частях компании — над приложениями этой общей теории.

Я рад переводу этой статьи на мой родной язык. Моя искренняя признательность Валерию Непомнящему, Павлу Емельянову и всем, кто участвовал в публикации этого сборника статей.

Юрий Гуревич
Редмонд, США
май 2004

1. ВВЕДЕНИЕ

В 1982 г. я пришел из математики в информатику. Изучая язык программирования, я заметил, что разными компиляторами он интерпретируется различно. Я заинтересовался, как и многие другие до меня, тем, что же в точности представляет собой семантика языка программирования? Что такое семантика конкретной программы? В этой связи я изучал знания и опыт того времени, в частности, денотационную семантику и алгебраические спецификации. Некоторые аспекты этих строгих подходов привлекли логика и бывшего алгебраиста. Тем не менее, я чувствовал, что ни один подход не является реалистичным.

Мне пришло в голову, что, в некотором смысле, Тьюринг решил проблему семантики программ. В то время как "официальный" тезис Тьюринга состоит в том, что каждая вычислимая функция является вычислимой по Тьюрингу, неформальный аргумент Тьюринга подтверждает в его пользу и более строгий тезис — каждый алгоритм может быть промоделирован некоторой машиной Тьюринга. В соответствии с этим строгим тезисом программа может быть промоделирована, и поэтому ей может

быть придан точный смысл посредством машины Тьюринга (МТ). На практике было бы смешно использовать МТ для того, чтобы представить программную семантику. МТ имеют дело с одиночными битами[2]. Обычно существует огромный разрыв между абстрактным уровнем программы и уровнем МТ. Даже если вы преуспеете в этой скучной работе по трансляции программы в язык МТ, вы не увидите за деревьями леса.

Возможно ли обобщить машины Тьюринга так, чтобы *любой* алгоритм, никогда не рассматривавшийся как абстрактный, мог быть промоделирован посредством обобщенной машины очень естественно и точно? Очевидным ответом представляется — нет. Мир алгоритмов чрезвычайно разнообразен, даже если ограничиться только последовательными алгоритмами, как я это сделал сейчас. Однако предположим, что такие обобщенные машины Тьюринга существуют. Что бы могло быть их состояниями? Огромный опыт математической логики указывает на то, что любой вид статической математической реальности может быть четко представлен как структура первого порядка. Таким образом, состояниями могли бы быть структуры первого порядка. Следующий вопрос: а какие инструкции следует иметь обобщенным машинам? Можно ли обойтись только ограниченным множеством инструкций? Здесь я сделал существенное наблюдение. Если мы стараемся сохранить один уровень абстракции (абстрагируясь от низкоуровневых деталей и не обращая внимания на возможную более общую картину) и если состояния алгоритма отражают всю необходимую информацию, то очень небольшое множество инструкций достаточно во всех случаях. Отсюда "Новый тезис" [18].

Я представил себе вычисление как некоторую эволюцию состояния. Удобно рассматривать отношения как специальные функции. Структуры первого порядка с чисто функциональным словарем называются алгебрами в универсальной алгебраической теории. Таким образом, новые машины были названы динамическими структурами, или алгебрами, или эволюционными алгебрами. Последнее название было некоторое время официальным, но затем сообщество эволюционных алгебр изменило его на машины абстрактных состояний или МАС.

Даже первые машины абстрактных состояний могли быть недетерминированными и могли взаимодействовать с окружением [19]. Затем были введены параллельные и мульти-агентные МАС [21]. Изначально МАС использовались для определения динамической семантики различных

[2]Очевидно, что не это свойство является главным недостатком МТ (особенно с учетом дальнейших рассуждений автора). Более существенным обстоятельством является последовательный доступ к памяти, отражающий тот факт, что данная модель реализует вычислительный процесс, выполняемый человеком (см. с. 12). — *Прим. перев.*

языков программирования. Позднее приложения разделились по многим направлениям [7]. Способность моделировать произвольные алгоритмы на их естественном уровне абстракции, без необходимости их реализации, делает МАС подходящими для высокоуровневого системного проектирования и анализа [8, 9]. Многочисленные свободно распространяемые приложения МАС можно найти на [1].

Читатель также найдет несколько статей по теории МАС в библиографии [7]. Одной из важных проблем, возникающих здесь, является независимость от формы представления данных. Рассмотрим, к примеру, графовые алгоритмы. Традиционные вычислительные модели [27] требуют, чтобы граф был представлен в той или иной форме, например, как строка или матрица смежности, даже в тех случаях, когда алгоритм не зависит от представления графа. Использование МАС (особенно параллельных МАС) позволяет программировать такие алгоритмы не зависимым от представления способом (в этой связи см. [3]).

Эта статья посвящена исходному МАС-тезису — *каждый последовательный алгоритм может быть пошагово промоделирован посредством некоторой подходящей последовательной МАС.* Статья о МАС-тезисе для параллельных вычислений в стадии подготовки[3]. Главная тема представленной статьи — формализация понятия последовательного алгоритма. Тезис Тьюринга был одной из основ (и одним из великолепных образцов) для этой работы. Другой основой было понятие структуры (первого порядка) [29]. Модель МАС — это попытка вложить динамику в понятие структуры.

По умолчанию в этой статье алгоритмы предполагаются последовательными, и в ней будут рассматриваться только последовательные МАС. На данный момент экспериментальные данные подтверждают этот тезис. Имеются также некоторые теоретические рассуждения, подтверждающие его. Они лишь упомянуты в литературе (см., например, [19]), но на протяжении многих лет они широко обсуждались в различных моих лекциях. Я пытался записать некоторые их этих объяснений в [23]. Данная работа — стройная журнальная версия рассуждений. Статья не предполагает знакомства с МАС.

Построение статьи

В разделе 2 дается краткая история проблемы формализации понятия последовательного алгоритма. В разделе 3, 4 и 5 формулируются соответственно постулаты Последовательного времени, Абстрактного состояния и Локальности решения. Обосновывается, что каждый последователь-

[3]На момент перевода эта статья [30] находилась в процессе публикации. — *Прим. перев.*

ный алгоритм удовлетворяет этим трем постулатам. В разделе 6 анализируется, что необходимо для того, чтобы моделировать произвольный последовательный алгоритм. Это приводит нас к понятию машины последовательных абстрактных состояний. Теорема, представленная в заглавии статьи, выводится из трех постулатов, упомянутых выше.

Раздел 7 содержит разного рода наблюдения относительно MAC. В разделе 8 Главная теорема обобщается на алгоритмы, взаимодействующие с их окружением. В разделе 9 обосновывается, что недетерминированные алгоритмы являются специальными интерактивными алгоритмами и поэтому наше обобщение на интерактивные алгоритмы покрывает и недетерминированные. Это требует небольшого изменения постулата Последовательного времени и некоторого обогащения языка программирования MAC. В приложении постулат Локальности решения выводится из утверждения (а также из постулатов Последовательного времени и Абстрактного состояния), являющегося, по-видимому, его более слабой версией.

Благодарности

Я чрезвычайно обязан сообществу MAC и особенно Андреасу Бласс у, Эгону Бёргеру и Деану Розенцвайгу. Продолжительные дискуссии в течение многих лет с Андреасом Блассом были необходимы для прояснения мыслей. Вклад Андреаса был отмечен уже в [18]. Эгон Бёргер сделал большой вклад в теорию и практику MAC. Большое количество MAC-приложений было разработано под его руководством. Деан Розенцвайг никогда не упускал возможности образно покритиковать. Большую пользу этой статье принесли также замечания Колина Кэмпбелла, Джима Хаггинса, Джима Каджия, Стивена Линделла, Льва Нахмансона, Питера Пеппингхауса, Григора Розу и Маргуса Веанеса. Редакторская работа Кшиштофа Апта была также, несомненно, очень полезной.

2. КРАТКАЯ ИСТОРИЯ ПРОБЛЕМЫ

Зачастую полагают, что проблема формализации понятия последовательного алгоритма была решена Чёрчем [11] и Тьюрингом [32]. Например, по Савэйджу [26], алгоритм — это вычислительный процесс, определяемый посредством машины Тьюринга. Но Чёрч и Тьюринг не решили проблему формализации понятия последовательного алгоритма. Вместо этого они предложили формализации (различные, но эквивалентные) понятия вычислимой функции. При этом имеется более чем один алгоритм, вычисляющий эту функцию.

Замечание 2.1. *Оба, и Чёрч, и Тьюринг, интересовались классической проблемой выполнимости (отыскание алгоритма, проверяющего выполнима ли данная формула первого порядка), являвшейся центральной проблемой математической логики в то время [6]. Они использовали формализацию понятия вычислимой функции для того, чтобы доказать неразрешимость классической проблемы выполнимости. В частности, Тьюринг выдвинул тезис, что каждая вычислимая функция вычислима посредством некоторой машины Тьюринга. Он доказал, что не существует машины Тьюринга, проверяющей выполнимость формулы первого порядка. В соответствии с тезисом проблема выполнимости не является вычислимой. Заметим, что формализация Чёрча—Тьюринга является либеральной — функция, вычислимая по Чёрчу—Тьюрингу, может быть невычислимой в любом практическом смысле. Однако их формализация допускает существование неразрешимых проблем.*

Конечно же, понятия алгоритма и вычислимой функции взаимосвязаны: по определению, вычислимая функция — это функция, вычислимая с помощью некоторого алгоритма. Оба, и Чёрч, и Тьюринг, говорили о такого рода алгоритмах. В соответствии со строгим тезисом Тьюринга, упомянутым во Введении, всякий алгоритм может быть промоделирован на машине Тьюринга. Более того, эксперты по сложности вычислений согласны, что любой алгоритм может быть промоделирован на машине Тьюринга с полиномиальным замедлением. Однако машина Тьюринга может работать слишком долго (при этом ее головка будет ползать взад и вперед по бесконечной ленте) для того, чтобы промоделировать один шаг данного алгоритма. Полиномиальное замедление может быть неприемлемым.

В то время как Тьюринг анализировал “человеческий” компьютер, Колмогоров и Успенский [14] пришли к их машинной модели (машины КУ), анализируя вычисления с точки зрения физики. Каждый бит информации должен быть представлен в физическом пространстве и может иметь только определенное количество соседних битов. Можно мыслить машину КУ как обобщенную машину Тьюринга, где лента — переконфигурируемый граф ограниченной степени. Машины Тьюринга не могут эффективно моделировать машины КУ [2]. Колмогоров и Успенский не сформулировали никакого тезиса. В статье [20], посвященной памяти Колмогорова, я попытался сделать это за них: “Всякие вычисления, выполняющие только одно ограниченное локальное действие за такт, могут рассматриваться (не только быть промоделированными, но фактически являться таковыми) как вычисления на подходящей машине КУ”. Успенский [33, с. 396] согласился.

Находясь под влиянием конвэевской "Игры Жизни", Ганди [15] привел доводы в пользу того, что тьюринговский анализ "человеческих" вычислений неприменим непосредственно к механическим устройствам. Наиболее важным является то, что механическое устройство может быть чрезвычайно параллельным. Ганди выдвинул четыре принципа, которым каждая такая машина должна удовлетворять. "Наиболее важный из них, носящий название «принцип локальной обусловленности», отвергает возможность мгновенного действия на расстоянии. Хотя эти принципы обосновываются посредством обращения к геометрии пространства–времени, формулировка является вполне абстрактной и может быть применима ко всем типам автоматов и алгебраических систем. Доказано, что если устройство удовлетворяет этим принципам, то его последовательные состояния образуют вычислимую последовательность." Работа Ганди вызвала интересную дискуссию в сообществе математической логики. Я обращусь к этим проблемам в будущей работе о MAC-тезисе для параллельных вычислений.

По-видимому, не зная о машинах КУ, Шёнхаге [28] определил машины, модифицирующие память, которые близки к машинам с указателями Кнута [25, с. 462–463]. Машинная модель Шёнхаге может рассматриваться как обобщение модели Колмогорова—Успенского, в которой граф ориентирован и только выходные степени вершин в нем ограничены. Это обобщение является естественным с точки зрения вычислений с указателями, которые используются на наших компьютерах. Однако с физической точки зрения это не столь естественно, так как вершина может быть напрямую доступна из неограниченного множества других вершин[4]. В любом случае уровень абстракции модели Шёнхаге выше, чем модели Колмогорова–Успенского. Неизвестно, возможно ли всякую машину Шёнхаге пошагово промоделировать на некоторой машине КУ.

Машины с произвольным доступом Кука и Рекхова [12] являются более мощными, чем машины Шёнхаге. Другие вычислительные модели можно найти в [27].

В приложениях алгоритм может использовать мощные операции (матричное умножение, дискретное преобразование Фурье и т.д.) как уже данные. На абстрактном уровне алгоритма такая операция исполняется за один шаг и проблемы, связанные с реальным исполнением операции, упрятаны в реализацию. Далее, состояние высокоуровневого алгоритма

[4]Можно задаться вопросом, насколько точно модель Колмогорова—Успенского соответствует физическим ограничениям. В конечном евклидовом пространстве объем сферы радиуса n ограничен полиномом от n. Соответственно, можно было бы ожидать полиномиальную оценку на количество вершин в любой окрестности радиуса n (в теоретико-графовом смысле) любого состояния данной машины КУ, однако в действительности такая окрестность может содержать экспоненциально большое количество вершин. Этот факт используется в статье Григорьева, упомянутой выше.

не обязано быть конечным (что противоречит первому принципу Ганди). Существует модель с высокоуровневыми абстракциями. Например, вычислительная модель Блюма, Шуба и Смэйла [5], оперирующая с вещественными числами произвольной точности. Высокоуровневые описания параллельных алгоритмов разработаны в [10].

Я искал машинную модель (с подходящим языком программирования) такую, что любой последовательный алгоритм, но, тем не менее, абстрактный, мог бы быть пошагово промоделирован некоторой машиной данной модели. Позвольте назвать такую модель универсальной по отношению к последовательным алгоритмам. Тьюринговская модель является универсальной по отношению к вычислимым функциям, но не по отношению к алгоритмам. По существу, тезис последовательных МАС состоит в том, что модель последовательных МАС является универсальной по отношению к последовательным алгоритмам. Мне не известна другая модель последовательных алгоритмов, которая была бы универсальной по отношению к последовательным алгоритмам в этом строгом смысле.

3. ПОСЛЕДОВАТЕЛЬНОЕ ВРЕМЯ

Это первый из трех разделов о формализации понятия последовательного алгоритма на фиксированном уровне абстракции. Здесь формулируется наш первый постулат.

3.1. Синтаксис

Допустим неформально, что любой алгоритм A может быть представлен посредством конечного текста, который объясняет алгоритм и не предполагает никакого специального знания. Например, если A дается программой на некотором языке программирования PL, то конечный текст должен объяснять соответствующую часть PL в дополнение к заданию PL-программы A. Мы не собираемся анализировать такого рода нотации. Уже имеется приводящее в замешательство многообразие языков программирования и различных других способов представления алгоритмов. Оставим в покое нотации, которые могли бы использоваться сейчас или в будущем. Сосредоточимся на поведении алгоритмов.

3.2. Поведение

Пусть A — последовательный алгоритм.

Постулат 1 (Последовательное время). *Алгоритму A соответствуют:*

– *множество* $\mathcal{S}(A)$, *элементы которого называются состояниями* A;
– *подмножество* $\mathcal{I}(A)$ *множества* $\mathcal{S}(A)$, *элементы которого называются начальными состояниями* A;
– *отображение* $\tau_A : \mathcal{S}(A) \to \mathcal{S}(A)$, *которое будет называться одношаговым преобразованием* A.

Эти три составляющие позволяют нам определить вычислительные последовательности алгоритма A.

Определение 3.1. *Последовательность (или вычисление) алгоритма* A *— это конечная или бесконечная последовательность состояний*

$$X_0, X_1, X_2, \ldots,$$

в которой $X_0 \in \mathcal{I}(A)$ *и каждое* $X_{i+1} = \tau_A(X_i)$.

Мы абстрагируемся от физического времени вычислений. Время вычислений, используемое в постулате Последовательного времени, можно было бы назвать логическим. Переход от X_0 к X_1 — это первый шаг вычислений, переход от X_1 к X_2 — это второй шаг вычислений и т.д. Шаги вычислений алгоритма A образуют последовательность. В этом смысле время вычислений является последовательным.

Определение 3.2. *Алгоритмы* A *и* B *— эквивалентны, если*

$$\mathcal{S}(A) = \mathcal{S}(B), \mathcal{I}(A) = \mathcal{I}(B) \text{ и } \tau_A = \tau_B.$$

Следствие 3.3. *Эквивалентные алгоритмы имеют одинаковые вычислительные последовательности.*

Другими словами, эквивалентные алгоритмы имеют одинаковое поведение. Мы рассматриваем алгоритмы с точностью до отношения эквивалентности. Поскольку поведение алгоритма определяется его тремя составляющими, не имеет значения, что представляет алгоритм сам по себе.

3.3. Дискуссия

3.3.1. Состояния

Для нас состояния — это мгновенные описания алгоритма. Однако имеется тенденция использовать термин “состояние” в более ограниченном смысле. Программист может говорить о начальных, ожидающих, активных и т.д. состояниях программной компоненты, даже если эти состояния не являются полными мгновенными описаниями компоненты. Вместо этого мы предпочитаем говорить о начальных, ожидающих, активных и т.д. *режимах* компоненты. Иногда говорят, что машины Тьюринга имеют лишь конечное число состояний. С нашей точки зрения,

множество состояний машины Тьюринга бесконечное. Конечным является управляющее устройство машины, которое имеет лишь конечное число конфигураций, но состояние машины включает не только текущую конфигурацию управляющего устройства, но также текущую конфигурацию ленты и текущую позицию головки чтения/записи.

Назовем состояние X алгоритма A *достижимым*, если X входит в некоторую последовательность A. Множество достижимых состояний A однозначно определяется $\mathcal{I}(A)$ и τ_A. Однако мы не предполагаем, что $\mathcal{S}(A)$ состоит из одних достижимых состояний. Интуитивно, $\mathcal{S}(A)$ — это множество *a priori* состояний алгоритма A. Зачастую оно проще, чем множество достижимых состояний, и поэтому более удобно для исследования. Например, состояние машины Тьюринга может задаваться *любой* строкой $uaqv$, где q — текущее состояние устройства управления, a — символ в текущей обозреваемой ячейке и u, v — строки в алфавите МТ такие, что лента — это строка uav, за которой следует бесконечный хвост пробелов. Не все из этих состояний достижимы в общем случае, и в действительности проблема достижимости состояний для машины Тьюринга может быть неразрешимой. С другой стороны, ограничение достижимыми состояниями было бы слишком строгим для целей данной работы.

В приложениях система может рассматриваться с различных точек зрения. Для этого могут использоваться различные уровни абстракции. Решение, что является состояниями системы, включает выбор одного их этих уровней абстракции.

3.3.2. Проблема остановки

Зачастую требуется, чтобы алгоритм останавливался на всех входах (см., например, [26]). Однако многие полезные алгоритмы не предполагают остановки. Одним из математических примеров является решето Эратосфена, которое выдает простые числа. Таким образом, мы не требуем, чтобы алгоритм останавливался на всех входах. Но, конечно же, конкретный алгоритм A может иметь конечные вычислительные последовательности. Мы полагаем, что $\tau_A(X) = X$, если A останавливается в X.

В качестве альтернативного решения, в дополнение к $\mathcal{S}(A)$, $\mathcal{I}(A)$ и τ_A мы могли бы определить множество $\mathcal{T}(A)$ заключительных состояний для алгоритма A. Было бы естественным ограничить τ_A на $\mathcal{S}(A) - \mathcal{T}(A)$. Мы не делаем этого, что упрощает определение активного окружения. Представим, что A переходит в состояние X, которое является заключительным по определению A, однако затем окружение изменяет состояние и вычисление продолжается. Было ли X действительно заключитель-

ным? Мы не рассматриваем эту проблему, так же как и проблему, касающуюся взаимосвязи заключительных и ошибочных состояний. Более того, удобно, чтобы преобразование τ_A являлось всюду определенным. Это невнимание к заключительным состояниям не существенно для теории MAC. Мы готовы сделать заключительные состояния явными, если это необходимо или удобно [4].

3.3.3. Холостые переходы

Может ли вычислительная последовательность иметь холостые переходы такие, что $X_{i+1} = \tau_A(X_i) = X_i$? Это не имеет значения для наших целей в данной статье. Правила холостых переходов играют незначительную роль в последовательных вычислениях. Случившись однажды, холостой переход будет повторяться до конца (если он существует) вычислительной последовательности. Для простоты мы исключаем здесь холостые переходы. Правила холостых переходов играют важную роль в мультиагентных вычислениях, но это другая история.

3.3.4. Эквивалентность

Можно сказать, что наше понятие эквивалентности алгоритмов является слишком сильным и что более слабое отношение эквивалентности могло бы быть более подходящим. Рассмотрим, например, два следующих простых алгоритма A и B. Каждый из них выполняет только два действия. A присваивает 1 переменной x и затем 2 переменной y, а B сначала присваивает 2 переменной y и затем 1 переменной x. В соответствии с нашим определением, A и B не являются эквивалентными. Имеет ли это смысл? Ответ зависит от целей. Наша цель — доказать, что для каждого алгоритма имеется эквивалентная машина абстрактных состояний. Сильнее эквивалентность — сильнее теорема. Более слабая эквивалентность также может быть использована; теорема останется истинной.

3.4. Что осталось?

В действительности то, что алгоритм является последовательным, означает больше, чем последовательное время. Имеется и нечто другое.

Рассмотрим, например, алгоритм достижимости в графе, который итеративно исполняет следующий шаг. Предполагается, что изначально только выделенная вершина `Source` удовлетворяет унарному отношению R.

```
do for all x, y with Edge(x, y) ∧ R(x) ∧ ¬R(y)
    R(y) := true
```

Алгоритм является последовательным по времени, но он чрезвычайно параллельный, не последовательный. Последовательный алгоритм должен делать только локальные изменения, и глобальное изменение должно быть ограниченным.

Требование ограниченного изменения также не является достаточным. Рассмотрим следующий графовый алгоритм, проверяющий, имеет ли заданный граф изолированные вершины.

```
if ∀x∃y Edge(x, y) then Output := false
                   else Output := true
```

Алгоритм изменяет только одно булевское значение, но на одном шаге использует граф целиком и поэтому не является последовательным. Последовательный алгоритм должен не только ограниченно изменять вычислительное состояние на одном шаге, но и использовать при этом информацию об ограниченной части состояния.

Некоторые исследователи находят удобным говорить в терминах действий. Заметим, что одношаговое преобразование алгоритма может состоять из нескольких различных действий. Например, машина Тьюринга может изменить состояние своего управляющего устройства, напечатать символ в текущую ячейку ленты и переместить свою головку; и все это на одном шаге. Верно ли, что всякое действие можно разделить на атомарные действия? Если да, то было бы разумно требовать, чтобы одношаговое преобразование состояло из ограниченного числа атомарных действий. Мы переходим к исследованию этих проблем.

4. АБСТРАКТНЫЕ СОСТОЯНИЯ

До сих пор состояния были просто элементами множества состояний. Для исследования проблем, возникших в предыдущем подразделе, рассмотрим понятие состояния более внимательно. Покажем, что состояния могут рассматриваться как структуры (первого порядка) математической логики. Это — часть нашего второго постулата, постулата Абстрактного состояния.

4.1. Структуры

Понятие структуры (первого порядка) приведено в учебниках по математической логике. Мы используем некоторую модификацию классического понятия [19].

4.1.1. Синтаксис

Словарь — это конечный набор функциональных символов, каждый из которых имеет фиксированное количество аргументов. Некоторые функциональные символы могут быть помечены как *предикаты*, или как статичные, или как и то, и другое. Всякий словарь содержит знак равенства, 0-местные имена `true`, `false`, `undef`, 1-местное имя `Boole` и имена обычных булевских операций. Все эти *логические символы* являются статическими, а за исключением `undef` — еще и предикатами.

Термы (более точно, *базовые* термы; по умолчанию в данной статье термы являются базовыми) определяются стандартным индуктивным способом. 0-местный функциональный символ — это терм. Если f — j-местный функциональный символ, где $j > 0$, и $t_1, \ldots, t_j$ — термы, то $f(t_1, \ldots, t_j)$ — термы. Если самый внешний функциональный символ является предикатом, то терм является *булевским*.

4.1.2. Семантика

Структурой X над словарем Υ является непустое множество S (*базовое множество* X) вместе с интерпретацией функциональных символов из Υ над S. Элементы S также называются элементами X. j-местный функциональный символ, интерпретируемый как функция из S^j в S, — *базовая функция* структуры X. Мы отождествляем 0-местную функцию с ее значением. Таким образом, в контексте данной структуры, `true` означает специальный элемент, а именно — интерпретацию символа `true`. То же самое применимо к `false` и `undef`. Требуется, чтобы интерпретация `true` отличалась от интерпретаций символов `false` и `undef`. Интерпретацией j-местного предиката R является функция из S^j в `{true,false}`. Это — *базовый предикат* структуры X. Знак равенства интерпретируется как отношение равенства на базовом множестве. Базовый предикат R можно мыслить как множество кортежей $\overline{a}$ таких, что $R(\overline{a}) =$ `true`. Если предикат R — унарный, то он может рассматриваться как *универсум*. `Boole` является (интерпретируется как) универсумом `{true,false}`. Булевские операции определены обычным образом над `Boole` и выдают `false`, если по крайней мере один из аргументов не является булевским. `undef` позволяет нам представлять интуитивно частичные функции как всюду определенные.

Замечание 4.1. *Можно оговорить в качестве особого условия, что* `undef` *является значением для булевских операций по умолчанию. Тогда они становятся частичными отношениями. Несмотря на то, что частичные отношения являются естественными, традиция математической логики — иметь дело с всюду определенными отношениями.*

При использовании других начальных положений теории, которые мы не рассматриваем в этой статье, можно иметь как классы всюду определенных, так и частичных отношений.

Прямая индукция дает значение $Val(t, X)$ терма t в структуре X, чей словарь включает словарь t. Если $Val(t, X) = Val(t', X)$, можно говорить, что $t = t'$ в X. Если $t = \texttt{true}$ (соответственно $t = \texttt{false}$) в X, можно говорить, что t выполнено или верно (соответственно не выполнено или неверно) в X.

4.1.3. Изоморфизм

Пусть X и Y — структуры с одним словарем Υ. Напомним, что *изоморфизм* X на Y — это взаимно-однозначная функция ζ из базового множества X на базовое множество Y такое, что $f(\zeta x_1, \ldots, \zeta x_j) = \zeta x_0$ в Y всякий раз, когда $f(x_1, \ldots, x_j) = x_0$ в X. Здесь f — j-местный символ из Υ.

4.2. Постулат Абстрактного состояния

Пусть A — последовательный алгоритм.

Постулат 2 (Абстрактное состояние).
- *Состояниями A являются структуры первого порядка.*
- *Все состояния A имеют один и тот же словарь.*
- *Преобразование τ_A сохраняет базовое множество состояния и все статические функции.*
- *$\mathcal{S}(A)$ и $\mathcal{I}(A)$ замкнуты при изоморфизмах. Более того, всякий изоморфизм из состояния X на состояние Y является также изоморфизмом из $\tau_A(X)$ на $\tau_A(Y)$.*

В оставшейся части раздела мы обсудим эти четыре части постулата.

4.3. Состояния как структуры

Огромный опыт математической логики и ее приложений показывает, что любая статическая математическая ситуация может быть точно описана как структура первого порядка. Удобно отождествлять состояния с соответствующими структурами. Базовые функции, которые могут меняться от одного состояния данного алгоритма к другому, называются *динамическими*. Многочисленные примеры состояний как структур могут быть найдены в литературе по МАС [7]. Здесь мы приведем лишь два простых примера.

4.3.1. Пример: машины Тьюринга

Состояние машины Тьюринга может быть формализовано следующим образом. Базовое множество структуры включает объединение непересекающихся универсумов

$$\texttt{Control} \cup \texttt{Alphabet} \cup \texttt{Tape},$$

которые также не пересекаются с множеством {`true, false, undef`}.

— `Control` является множеством (или представляет множество) состояний конечного устройства управления. Каждый элемент `Control` выделен, т.е. имеет соответствующий (0-местный функциональный) символ в словаре. Кроме того, имеется динамический выделенный элемент `CurrentControl`, который "живет" в `Control`. Другими словами, мы имеем 0-местный функциональный символ `CurrentControl`, чье значение может меняться от одного состояния к другому.

— `Alphabet` — это алфавит ленты. Каждый элемент `Alphabet` также выделен. Один из этих элементов, `Blank`, называется пробелом.

— В качестве `Tape` может быть взято множество целых чисел (представляющих ячейки ленты). Для него определены унарные операции `Successor` и `Predecessor`. Для простоты мы предполагаем, что лента бесконечна в обоих направлениях. Также имеется динамический 0-местный функциональный символ `Head`, принимающий значения в `Tape`.

И, наконец, мы имеем унарную функцию

$$\texttt{Content} : \texttt{Tape} \to \texttt{Alphabet},$$

которая присваивает всем ячейкам (но конечному их числу) `Blank` и каждому элементу, расположенному вне `Tape`, `undef`.

Следующее очевидное правило представляет инструкцию машины Тьюринга:

```
if CurrentControl = q1 and Content(Head) = σ1 then
   do-in-parallel
      CurrentControl := q2
      Content(Head) := σ2
      Head := Successor(Head)
```

Вся целиком программа машины Тьюринга может быть записана как некоторый `do-in-parallel` блок правил, подобных этому. Впоследствии `do-in-parallel` сокращается до `par`.

4.3.2. Пример: алгоритм Евклида

Алгоритм Евклида вычисляет наибольший общий делитель d двух заданных натуральных чисел a и b. Один его шаг может быть описан следующим образом:

```
if      b = 0 then d := a
else if b = 1 then d := 1
else
   par
      a := b
      b := a mod b
```

Базовое множество любого состояния X алгоритма включает множество натуральных чисел, содержащее 0, 1 в качестве различных элементов и бинарную операцию `mod`. Кроме того, имеются три различных динамических элемента a, b и d. Если $a = 12$, $b = 6$, $d = 1$ в X, то $a = 6$, $b = 0$, $d = 1$ в следующем состоянии X' и $a = 6$, $b = 0$, $d = 6$ в следующем состоянии X''.

4.3.3. Структуры с точки зрения логика

Для логика структуры, введенные в этом разделе, являются структурами первого порядка. В логике имеются и другие структуры, например, второго или высших порядков. Иногда они могут быть более подходящими для представления состояний. Рассмотрим, например, графовый алгоритм, манипулирующий не только отдельными вершинами, но и множествами вершин. В этом случае подходят структуры второго порядка. Однако структуры второго порядка, высших порядков и т.д. могут рассматриваться как специальные структуры первого порядка. В частности, графовые структуры второго порядка, необходимые для упомянутого графового алгоритма, могут рассматриваться как двусортные первого порядка, в которых элементами первого сорта являются вершины, а второго — множества вершин. В добавление к отношению на элементах первого сорта, определяющему ребра, имеется также межсортное бинарное отношение $\epsilon(x, y)$, выражающее то, что вершина x принадлежит множеству вершин y.

Термин “структура первого порядка” может ввести в заблуждение. Он отражает тот факт, что обсуждаемые структуры используются для определения семантики логики первого порядка. В случае двусортных графовых структур, упомянутых выше, не существует высказывания первого порядка, выражающего то, что всякое множество вершин представляется некоторым элементом второго сорта. Но это не имеет значения для наших целей. Мы используем структуры первого порядка, не ограничивая себя

логикой первого порядка. Это общая практика в математике; вспомните, например, теорию графов, теорию групп или теорию множеств.

4.4. Фиксированный словарь

В логике словари необязательно являются конечными. В нашем случае, по определению, словари конечные. Это соответствует неформальному предположению, что программа алгоритма A представляется конечным текстом.

Выбор словаря диктуется выбранным уровнем абстракции. При точной формализации словарь отражает скорее действительно инвариантные особенности алгоритма, чем детали отдельного состояния. В частности, словарь не меняется в процессе вычислений. Можно представлять вычисление как некоторую эволюцию начального состояния. Словарь не меняется в процессе этой эволюции.

Является ли обоснованным требование неизменности словаря в процессе эволюции? Можно представить себе алгоритм, требующий все больше и больше функций или предикатов во время своей работы. Например, алгоритм раскраски графов может требовать все больше цветов для бóльших графов. Естественно представлять цвета как унарные отношения. Заметим, однако, что конечная программа алгоритма раскраски должна обеспечивать систематический способ работы с цветами. Например, цвета могут образовывать расширяемый массив неограниченной длины. Математически отсюда возникает *бинарное* отношение $C(i, x)$, для которого множеством вершин i-го цвета является $\{x : C(i, x)\}$. В общем случае, если алгоритму необходимо все больше и больше j-местных функций некоторого вида, то необходимость в $(j + 1)$-местной функции становится реальной. Возможно, что в качестве альтернативы было бы удобнее рассматривать эти j-местные функции элементами специального универсума.

Имеются также так называемые самомодифицирующие "не фон-неймановские" алгоритмы, которые изменяют свои программы в процессе вычислений. Для такого алгоритма так называемая "программа" является просто частью данных. Собственно программа меняет часть данных, а сама при этом остается неизменной.

4.5. Неизменяемое базовое множество

Несмотря на то, что базовое множество может меняться от одного начального состояния к другому, оно неизменно в процессе вычислений. Все состояния данной вычислительной последовательности имеют одно и то же базовое множество. Правдоподобно ли это? Имеются, например,

графовые алгоритмы, которые требуют добавления новых вершин в текущий граф. Но откуда берутся новые вершины? Мы можем формализовать кусочек внешнего мира и оговорить в качестве особого условия, что начальное состояние содержит бесконечное "голое" множество — резерв. Новые вершины берутся из резерва, и поэтому базовое множество не изменяется в процессе эволюции.

Кто выдает элементы из резерва? Окружение. В приложении программа может использовать некоторую форму команды NEW; операционная система будет обязана предоставить больше памяти. Формализуя это, мы можем использовать специальную *внешнюю* функцию для получения элемента из резерва. Она является внешней в том смысле, что контролируется окружением.

Даже если интуитивное начальное состояние может быть конечным, бесконечно много дополнительных элементов пробили свою дорогу в начальную структуру только потому, что они, возможно, станут нужными позднее. Обосновано ли это? Я думаю, что да. Конечно же, мы можем оставить идею о неизменяемом базовом множестве и импортировать новые элементы из внешнего мира. Вероятно, концептуально это не имеет значения. С технической точки зрения более удобно иметь кусочек внешнего мира внутри состояния.

Это не первый случай, когда мы отражаем кусочек внешнего мира внутрь структуры. Мы предполагали, что структура содержала (денотаты) `true` и `false`, которые позволяли нам представлять предикаты как булевозначные функции. Интуитивное состояние могло бы иметь предикаты и не содержать их значения; вспомните граф в качестве примера. Подобным образом мы предполагали, что структура содержала `undef`, который позволял нам представлять интуитивно частичные функции как всюду определенные.

4.6. Абстракция с точностью до изоморфизма

Последняя часть постулата Абстрактного состояния состоит из двух утверждений. Она отражает тот факт, что мы работаем на фиксированном уровне абстракции. Структура должна рассматриваться лишь как представление класса изоморфных ей структур. Только этот класс имеет значение. Отсюда первое из двух утверждений: различные изоморфные структуры — просто различные представители одного и того же класса, и если одна из них является состоянием данного алгоритма A, то и другая также должна быть состоянием этого алгоритма[5]. То, как дан-

[5]Специалист по теории множеств может указать, что это требует наличия собственного класса состояний, потому что любое состояние имеет собственный класс изоморфных копий. Проблему можно было бы обойти, зафиксировав некоторое об-

ное состояние представляет свой класс изоморфных состояний, не имеет значения. Если она существует, то текущий уровень абстракции был выбран плохо. Значимые детали должны быть сделаны явными. Словарь и базовые функции должны быть пересмотрены.

Для того чтобы рассмотреть второе утверждение, предположим, что X и Y — различные состояния алгоритма A и ζ — изоморфизм из X на Y. ζ отображает базовое множество X на базовое множество Y и сохраняет все функции X. Например, если $f(a) = b$ в X, то $f(\zeta a) = \zeta b$ в Y. Поскольку базовое множество неизменно, базовые множества $\tau_a(X)$ и $\tau_a(X)$ являются таковыми соответственно для X и Y. Вопрос, сохраняет ли ζ функции $\tau_A(X)$? Посмотрим на Y просто как на другое представление X. Элемент x из X представлен посредством элемента ζx из Y. Но представление состояния не должно иметь никакого значения. Для того чтобы продолжить этот пример, предположим, что τ_A отображает $f(a)$ в $c \in X$ так, что $f(a) = c$ в $\tau_A(X)$. В ζ-представлении X (т.е. в Y) τ_A отображает $f(\zeta a)$ в ζc так, что $f(\zeta a) = \zeta c$ в $\tau_A(Y)$.

5. ЛОКАЛЬНОСТЬ ПРИНЯТИЯ РЕШЕНИЯ

5.1. Состояния как вид памяти

Удобно рассматривать структуру X как память специального вида. Если f — j-местный функциональный символ и $\overline{a}$ — кортеж j элементов из X, то пара $(f,\ \overline{a})$ — *элемент памяти*. $Content_X(f,\ \overline{a})$ — элемент $f(\overline{a})$ в X.

Если $(f,\ \overline{a})$ — элемент памяти из X и b — элемент X, то $(f,\ \overline{a},\ b)$ — *обновление* состояния X. Обновление $(f,\ \overline{a},\ b)$ является *тривиальным*, если $b = Content_X(f,\ \overline{a})$. Для того чтобы выполнить обновление $(f,\ \overline{a},\ b)$, необходимо заменить содержимое элемента памяти $(f,\ \overline{a})$ на b.

Два обновления *конфликтуют*, если они обращаются к одному и тому же элементу памяти, но являются различными. Множество обновлений *совместимо*, если оно не имеет конфликтующих обновлений. Для того чтобы исполнить совместимое множество обновлений, нужно исполнить все обновления этого множества одновременно. В случае несовместимого множества обновлений ничего не делается. Результат исполнения множества обновлений Δ над X будет обозначаться $X + \Delta$.

Лемма 5.1. *Если X, Y — структуры с одними и теми же словарем и базовым множеством, то существует единственное совместимое множество Δ нетривиальных обновлений X такое, что $Y = X + \Delta$.*

ширное множество и рассматривая только те структуры, чьи элементы принадлежат этому множеству. В качестве альтернативы можно разрешить $\mathcal{S}(A)$ и $\mathcal{I}(A)$ быть собственными классами. Мы будем просто игнорировать эту проблему.

Доказательство. X и Y имеют одни и те же элементы памяти. Требуемое множество Δ определяется следующим образом:

$$\Delta = \{\ (f,\ \overline{a},\ b) : b = \mathit{Content}_Y(f,\ \overline{a}) \neq \mathit{Content}_X(f,\ \overline{a})\}\,.$$

■

Множество Δ будет обозначаться $Y - X$.

5.2. Множество обновлений алгоритма в заданном состоянии

Пусть X — некоторое состояние алгоритма A. Из постулата Абстрактного состояния следует, что X и $\tau_A(X)$ имеют одни и те же элементы и элементы памяти. Определим

$$\Delta(A, X) \rightleftharpoons \tau_A(X) - X$$

так, что $\tau_A(X) = X + \Delta(A, X)$.

Лемма 5.2. *Предположим, что ζ — изоморфизм из состояния $X \in A$ на состояние $Y \in A$, и расширим очевидным образом ζ так, чтобы его область содержала как кортежи элементов, так и кортежи элементов памяти, обновлений и множеств обновлений состояния X. Тогда $\Delta(A, Y) = \zeta(\Delta(A, X))$.*

Доказательство. Использовать последнюю часть (абстракция с точностью до изоморфизма) постулата Абстрактного состояния. ■

5.3. Принцип Доступности

По умолчанию, в данной статье термы являются базовыми (т.е. не содержат переменных), но этот подраздел — исключение.

В соответствии с постулатом Абстрактного состояния алгоритм A не различает изоморфные состояния. Состояние X алгоритма A является просто частной реализацией его изоморфного класса. Как A может получить доступ к элементу a множества X? Один из способов — создать базовый терм, вычисление которого дает $a \in X$. Утверждение, что это — единственный способ, может быть названо *принципом Последовательного доступа.*

Можно представить другие способы доступа алгоритма A к элементу a вычислительного состояния. Например, могли бы существовать булевские термы ϕ и $\psi(x)$ такие, что ϕ — базовый, x — единственная свободная переменная в $\psi(x)$ и уравнение $\psi(x) =$ `true` имеет единственное решение в любом состоянии X, удовлетворяющем ϕ. Если эта информация доступна алгоритму A, он может вычислить ϕ в данном состоянии X

и затем, если ϕ верно в X, указать единственное решение a уравнения $\psi(x) = \texttt{true}$, предъявив терм $\psi(x)$. Описанная процедура использует волшебный скачок от булевского терма $\psi(x)$ к элементу a. Для того чтобы объяснить волшебство, введем новый 0-местный функциональный символ c для единственного решения $\psi(x) = \texttt{true}$, в случае, если ϕ верно. В противном случае c может равняться `undef`. Если мы разрешаем ϕ и $\psi(x)$ иметь параметры, нам понадобится ввести новый функциональный символ положительной местности. Это ведет к точной формализации данного алгоритма, который удовлетворяет принципу Последовательного доступа.

Определение 5.3. *Элемент a структуры X является доступным, если $a = Val(t,\ X)$ для некоторого базового терма t в словаре X. Элемент памяти $(f,\ \overline{a})$ является доступным, если каждый элемент кортежа $\overline{a}$ доступен. Обновление $(f,\ \overline{a},\ b)$ является доступным, если и $(f,\ \overline{a})$ и b — доступны.*

Принцип доступности и неформальное предположение, что всякий алгоритм имеет конечную программу, означает, что всякий конкретный алгоритм A использует в любом состоянии только ограниченное количество элементов. В самом деле, каждый элемент, используемый A, должен быть поименован некоторым базовым термом, но конечная программа может ссылаться только на конечное число базовых термов. Это не является формальным доказательством, и мы не собираемся анализировать синтаксис программ. По этой причине мы не повышаем принцип Доступности до статуса постулата, но он является движущей силой постулата Локальности решения.

5.4. Постулат Локальности решения

Будем говорить, что две структуры X и Y в одном и том же словаре X *совпадают* над множеством T Υ-термов, если $Val(t, X) = Val(t, Y)$ для всех $t \in T$. *Словарем* алгоритма называется словарь его состояний. Пусть A — последовательный алгоритм.

Постулат 3 (Локальности решения). *Существует конечное множество термов T в словаре алгоритма A такое, что если состояния X, Y алгоритма A совпадают над T, то $\Delta(A, X) = \Delta(A, Y)$.*

Интуитивно алгоритм A использует только часть данного состояния, которая определяется термами из T. Само по себе множество T — *свидетель локальности решения* для алгоритма A.

Пример. Нелогическая часть словаря алгоритма A состоит из 0-местного функционального символа f, унарного предиката P и унарного функционального символа S.

Каноническое состояние A состоит из множества натуральных чисел и трех дополнительных различных элементов (именуемых) `true, false, undef`. S — функция, выдающая следующий элемент натурального ряда для данного натурального числа. P — подмножество натуральных чисел. f вычисляет некоторое натуральное число.

Произвольное состояние A изоморфно одному из канонических состояний. Каждое состояние A является начальным. Пошаговое преобразование задается следующей программой:

`if` $P(f)$ `then` f `:=` $S(f)$

Ясно, что A удовлетворяет постулатам Последовательного времени и Абстрактного состояния. Для того чтобы показать, что он удовлетворяет постулату Локальности решения, нам нужно предъявить свидетеля локальности решения. Может показаться, что множество $T_0 \rightleftharpoons \{f, P(f), S(f)\}$ является таковым для A, но это не так. В самом деле, пусть X — каноническое состояние A, в котором верно $f = 0$ и $P(0)$. Положим $a \rightleftharpoons \mathit{Val}(\mathtt{true}, X)$ и $b \rightleftharpoons \mathit{Val}(\mathtt{false}, X)$ так, что $\mathit{Val}(P(0), X) = \mathit{Val}(\mathtt{true}, X) = a$. Пусть Y — состояние, полученное их X заменой интерпретации `true` на b и `false` на a так, что $\mathit{Val}(\mathtt{true}, Y) = b$ и $\mathit{Val}(\mathtt{false}, Y) = a$. Значение $P(0)$ не изменилось: $\mathit{Val}(P(0), Y) = a$, так что $P(0)$ ложно в Y. Тогда X, Y совпадают над T_0, но

$$\Delta(X, A) \neq \emptyset = \Delta(Y, A).$$

Множество $T = T_0 \cup \{\mathtt{true}\}$ является свидетелем локальности решения для A[6].

6. АНАЛИЗ ПОСТУЛАТОВ, МАШИНЫ АБСТРАКТНЫХ СОСТОЯНИЙ И ОСНОВНАЯ ТЕОРЕМА

До сих пор понятие последовательного алгоритма было неформальным. Сейчас мы готовы его формализовать.

Определение 6.1. *Последовательный алгоритм — это объект A, который удовлетворяет постулатам Последовательного времени, Абстрактного состояния и Локальности решения.*

Проанализируем произвольный алгоритм A. Пусть Υ — словарь A. В этом разделе все структуры являются Υ-структурами и все состояния являются состояниями A.

[6] В связи с этим примером мой коллега Лев Нахмансон рассказал мне следующую историю. “В моей предыдущей компании Текноматикс одна очень старая библиотека перестала работать после перехода на новую операционную систему. Нам потребовалось несколько дней для того, чтобы понять, что случилось. Макрос, сообщающий об успешном исполнении C-функции, был определен равным 0. Во включаемых файлах новой операционной системы он был переопределен на 1.”

Пусть T — свидетель локальности решения для A. Не уменьшая общности, предположим следующее:

— T замкнуто относительно подтермов: если $t_1 \in T$ и t_2 — подтерм t_1, то $t_2 \in T$.

— T содержит логические термы `true`, `false` и `undef`.

Назовем термы из T *критическими*. Для каждого состояния X значения критических термов из X будут называться *критическими элементами* X.

Лемма 6.2. *Если* $(f, (a_1, \ldots, a_j), a_0)$ *— обновление из* $\Delta(A, X)$, *то все элементы* $a_0, \ldots, a_j$ *являются критическими элементами* X.

Доказательство. Предположим противное: некоторое a не является критическим. Пусть Y — структура, изоморфная X, полученная из X заменой a_i новым элементом b. Из постулата Абстрактного состояния Y — состояние. Проверим, что $Val(t, Y) = Val(t, X)$ для каждого критического терма t. Ввиду выбора T $\Delta(A, Y)$ равно $\Delta(A, X)$ и, следовательно, содержит обновление $(f, (a_1, \ldots, a_j), a_0)$. Но $a_i \notin Y$. Также, из постулата Абстрактного состояния (его части, касающейся неизменяемости базового множества) $a_i \notin \tau_A(X)$. Следовательно, он не может принадлежать $\Delta(A, Y) = \tau_A(Y) - Y$. Противоречие. ■

Поскольку множество критических термов не зависит от X, размер $\Delta(A, X)$ ограничен. Таким образом, A — ограниченно изменяемое. Более того, всякое обновление в $\Delta(A, X)$ является атомарным действием. (В самом деле, словарь и базовое множество состояния не изменяются в процессе вычисления. Это происходит только с базовыми функциями. Для того чтобы изменить состояние минимально возможным способом так, чтобы результат был допустимой структурой, нужно изменить одну базовую функцию в одном месте, т.е. изменить содержимое одного элемента памяти. Это в точности то, что делает одно обновление.) Поэтому $\Delta(A, X)$ состоит из ограниченного количества атомарных действий.

Для того чтобы представить программным способом отдельные обновления $\Delta(A, X)$, введем правила обновления.

Определение 6.3. *Правило обновления словаря* Υ *имеет вид*

$$f(t_1, \ldots, t_j) := t_0\,,$$

где f *(динамический, т.е. не помечен как "статический")— j-местный функциональный символ из* Υ *и* $t_0, \ldots, t_j$ *— термы над* Υ. *Для того чтобы применить правило обновления над* Υ*-структурой* X, *нужно вычислить элементы* $a_i = Val(t_i, X)$ *и затем выполнить обновление* $(f, (a_1, \ldots, a_j), a_0)$ *над* X.

Определение 6.4. *Если* k *— произвольное натуральное число и* $R_1, \ldots, R_k$ *— правила над словарем* Υ, *то*

```
par
    R_1
    R_2
    ⋮
    R_k
endpar
```

является правилом в словаре Υ. *Для того чтобы применить* `par`*-правило к* Υ*-структуре* X*, нужно одновременно применить подправила* $R_1, \ldots, R_k$.

`par`-Правила называются *блоками*. Пустой блок (с пустым множеством подправил) сокращается до `skip`. Для программирования множества обновлений $\Delta(A, X)$ нам понадобятся только блоки с подправилами, соответствующими обновлениям, но дополнительное обобщение определения 6.4 будет для нас полезно. Для придания более строгой семантики правилам мы определим множество обновлений $\Delta(R, X)$, которое порождается правилом R в словаре Υ в любой Υ-структуре X.

Определение 6.5. *Если* R *— правило обновления* $f(t_1, \ldots, t_j) := t_0$ *и* $a_i = \mathrm{Val}(t_i, X)$ *для* $i = 0, \ldots, j$, *то*

$$\Delta(R, X) \rightleftharpoons \{(f, (a_1, \ldots, a_j), a_0)\}.$$

Если R *—* `par`*-правило с подправилами* $R_1, \ldots, R_k$, *то*

$$\Delta(R, X) \rightleftharpoons \Delta(R_1, X) \cup \cdots \cup \Delta(R_k, X).$$

Следствие 6.6. *Для каждого состояния* X *существует правило* R^X *такое, что*

1. R^X *использует только критические термы и*

2. $\Delta(R^X, X) = \Delta(A, X)$.

Далее в этом разделе R^X определено, как в следствии 6.6.

Лемма 6.7. *Если состояния* X *и* Y *совпадают над множеством критических термов* T, *то* $\Delta(R^X, Y) = \Delta(A, Y)$.

Доказательство. Имеем, что

$$\Delta(R^X, Y) = \Delta(R^X, X) = \Delta(A, X) = \Delta(A, Y).$$

Первое равенство верно потому, что R^X включает только критические термы, а также потому, что они имеют одни и те же значения в X и Y. Второе равенство верно по определению R^X. Третье — ввиду выбора T, а также потому, что X и Y совпадают над T. ■

Лемма 6.8. *Предположим, что X, Y — состояния и $\Delta(R^X, Z) = \Delta(A, Z)$ для некоторого состояния Z, изоморфного Y. Тогда $\Delta(R^X, Y) = \Delta(A, Y)$.*

Доказательство. Пусть ζ — изоморфизм из Y на подходящее Z. Расширим ζ на кортежи, элементы памяти, обновления и множество обновлений. Легко проверить, что $\zeta(\Delta(R^X, Y)) = \Delta(R^X, Z)$. Ввиду выбора Z $\Delta(R^X, Z) = \Delta(A, Z)$. Из леммы 5.2 $\Delta(A, Z) = \zeta(\Delta(A, Y))$. Поэтому $\zeta(\Delta(R^X, Y)) = \zeta(\Delta(A, Y))$. Осталось применить ζ^{-1} к обеим частям последнего равенства. ■

В каждом состоянии X отношение равенства между критическими элементами индуцирует отношение эквивалентности

$$E_X(t_1, t_2) \rightleftharpoons \mathit{Val}(t_1, X) = \mathit{Val}(t_2, X)$$

над критическими термами. Назовем состояния X, Y *T-подобными*, если $E_X = E_Y$.

Лемма 6.9. *$\Delta(R^X, Y) = \Delta(A, Y)$ для всякого состояния Y, T-подобного X.*

Доказательство. По лемме 6.8 достаточно найти состояние Z, изоморфное Y с $\Delta(R^X, Z) = \Delta(A, Z)$.

Во-первых, рассмотрим специальный случай, когда Y не пересекается с X, т.е. когда X и Y не имеют общих элементов. Пусть Z — структура, изоморфная Y, которая получена из Y заменой $\mathit{Val}(t, Y)$ на $\mathit{Val}(t, X)$ для всех критических термов t. (Определение структуры Z непротиворечиво: если t_1, t_2 — критические термы, то

$$\mathit{Val}(t_1, X) = \mathit{Val}(t_2, X) \iff \mathit{Val}(t_1, Y) = \mathit{Val}(t_2, Y),$$

так как X и Y являются T-подобными.) Из постулата Абстрактного состояния Z — состояние. Так как X и Z совпадают над T, лемма 6.7 дает $\Delta(R^X, Z) = \Delta(A, Z)$.

Во-вторых, рассмотрим общий случай. Заменим каждый элемент Y, принадлежащий X, на новый элемент. Это дает структуру Z, которая изоморфна Y и не пересекается с X. Из постулата Абстрактного состояния Z — состояние. Поскольку Z изоморфно Y, оно T-подобно Y и поэтому T-подобно X. Учитывая первую часть этого доказательства, $\Delta(R^X, Z) = \Delta(A, Z)$. ■

Для всякого состояния X существует булевский терм ϕ^X, который вычисляет `true` в структуре Y тогда и только тогда, когда Y T-подобен X. Требуемый терм утверждает, что отношение равенства на критических

термах является в точности отношением эквивалентности E_X. Поскольку имеется только конечное количество критических термов, то имеется только конечное количество возможных отношений эквивалентности E_X. Следовательно, имеется конечное множество состояний $\{X_1, \ldots, X_m\}$ таких, что всякое состояние T-подобно одному из состояний X_i.

Для того чтобы запрограммировать алгоритм A на всех состояниях, необходимо единственное правило, применимое ко всякому состоянию X и имеющее эффект R^X на X. Это естественным образом приводит к `if-then-else` конструкции и условным правилам.

Определение 6.10. *Если ϕ — булевский терм в словаре Υ и R_1, R_2 — правила в словаре Υ, то*

```
if φ
   then R1
   else R2
endif
```

является правилом в словаре Υ. Для того чтобы исполнить R над любой Υ-структурой X, необходимо вычислить ϕ над X. Если результат `true`*, то $\Delta(R, X) = \Delta(R_1, X)$, иначе $\Delta(R, X) = \Delta(R_2, X)$.*

`else`-Часть может быть опущена, если R_2 является `skip`. Такие `if-then` правила были бы достаточны для наших задач здесь, но дополнительная общность будет также полезна. В этой статье мы, как правило, будем опускать ключевые слова `endpar` и `endif`.

Последовательная MAC-*программа* Π в словаре Υ — это просто правило в словаре Υ. Таким образом, $\Delta(\Pi, X)$ корректно определено для всякой Υ-структуры X.

Лемма 6.11 (Основная лемма). *Для всякого последовательного алгоритма A в словаре Υ имеется MAC-программа Π в словаре Υ такая, что $\Delta(\Pi, X) = \Delta(A, X)$ для всех состояний X алгоритма A.*

Доказательство. Пусть $X_1, \ldots, X_m$ определены так же, как в обсуждении, которое следовало за леммой 6.9. Требуемая Π определяется следующим образом:

```
par
    if  φ^X1  then  R^X1
    if  φ^X2  then  R^X2
    ...
    if  φ^Xm  then  R^Xm
```

Лемма доказана. ■

Вложенные `if-then-else` правила дают альтернативное доказательство Основной леммы. Требуемой Π могла бы быть

```
if          φ^{X_1}    then    R^{X_1}
else if     φ^{X_2}    then    R^{X_2}
...
else if     φ^{X_m}    then    R^{X_m}
```

Если дана программа Π, определим

$$\tau_\Pi(X) \rightleftharpoons X + \Delta(\Pi, X).$$

Определение 6.12. *Последовательная машина абстрактных состояний B над словарем Υ определяется*

- *программой* Π *над словарем* Υ,
- *множеством* $\mathcal{S}(B)$ Υ*-структур, замкнутых относительно изоморфизмов и отображения* τ_Π,
- *подмножеством* $\mathcal{I}(B) \subseteq \mathcal{S}(B)$, *замкнутым относительно изоморфизмов,*
- *отображением* τ_B, *которое является ограничением* τ_Π *на* $\mathcal{S}(B)$.

Легко видеть, что машина абстрактных состояний удовлетворяет постулатам Последовательного времени, Абстрактного состояния и Локальности решения. По определению 6.1 — это алгоритм. Вспомним определение эквивалентных алгоритмов, определение 3.2.

Теорема 6.13 (Основная теорема). *Для всякого последовательного алгоритма A существует эквивалентная последовательная машина абстрактных состояний B.*

Доказательство. Ввиду Основной леммы, существует MAC-программа Π такая, что $\Delta(\Pi, X) = \Delta(A, X)$ для всех состояний X алгоритма A. Положим $\mathcal{S}(B) = \mathcal{S}(A)$ и $\mathcal{I}(B) = \mathcal{I}(A)$. ■

7. ЗАМЕТКИ О МАШИНАХ АБСТРАКТНЫХ СОСТОЯНИЙ

7.1. Конструктивность

Традиционно в работах по основаниям математики требуется, чтобы входы алгоритма являлись конструктивными объектами (обычно строками) и чтобы состояния алгоритма имели конструктивные представления. Одним из поборников этой традиции был Марков [24]. Как мы упоминали в разд. 2, эти требования конструктивности могут быть слишком ограничительными в приложениях, особенно для высокоуровневого конструирования и спецификации. Мы абстрагируемся от ограничений конструктивности. MAC являются алгоритмами, которые трактуют свои состояния как базы данных или как оракулы. В этом, очень практичном, смысле MAC являются исполняемыми. Большое количество свободно распространяемых инструментов для исполнения MAC могут быть найдены на

[1]. Абстрагирование от требований конструктивности внесло большой вклад как в простоту определения MAC, так и в их применимость.

7.2. Дополнительные примеры MAC-программ

Две MAC-программы были даны ранее в разд. 4. Здесь приведены три дополнительных примера.

7.2.1. Максимальная интервальная сумма

Следующая задача и ее математическое решение позаимствованы из [16]. Предположим, что A — функция из $\{0,1,\ldots,n-1\}$ в вещественные числа и $i,j,k \in \{0,1,\ldots,n\}$. Для всех $i \le j$ пусть $S(i,j) \rightleftharpoons \sum\limits_{i \le k < j} A(k)$. В частности, всегда $S(i,i) = 0$. Задача состоит в том, чтобы вычислить эффективно $S \rightleftharpoons \max\limits_{i \le j} S(i,j)$. Определим $y(k) \rightleftharpoons \max\limits_{i \le j \le k} S(i,j)$. Тогда $y(0) = 0$, $y(n) = S$ и

$$\begin{aligned} y(k+1) &= \max\{\max_{i \le j \le k} S(i,j), \max_{i \le k+1} S(i,k+1)\} = \\ &= \max\{y(k), x(k+1)\}, \end{aligned}$$

где $x(k) \rightleftharpoons \max\limits_{i \le k} S(i,k)$ так, что $x(0) = 0$ и

$$\begin{aligned} x(k+1) &= \max\{\max_{i \le k} S(i,k+1), S(k+1,k+1)\} = \\ &= \max\Big\{\max_{i \le k}\Big(S(i,k) + A(k)\Big), 0\Big\} = \\ &= \max\Big\{\Big(\max_{i \le k} S(i,k)\Big) + A(k), 0\Big\} = \\ &= \max\{x(k) + A(k), 0\}. \end{aligned}$$

Поскольку $y(k) \ge 0$, мы имеем

$$y(k+1) = \max\{y(k), x(k+1)\} = \max\{y(k), x(k) + A(k)\}.$$

Предположим, что 0-местные динамические функции k, x, y равны нулю в начальном состоянии. Требуемым алгоритмом будет

```
if k ≠ n then
   par
      x := max{x + A(k), 0}
      y := max{y, x + A(k)}
      k := k + 1
else S := y
```

7.2.2. Высокоуровневый взгляд на машины Тьюринга

Пусть `Alphabet`, `Control`, `Tape`, `Head` и `Content` такие же, как в примере о машинах Тьюринга из разд. 4. Удалим `Successor` и `Predecessor`. Вместо этого введем субуниверсум `Displacement` $= \{-1, 0, 1\}$ универсума `Tape`, каждый элемент которого различен, и бинарную функцию $+ :$ `Tape` $\times$ `Displacement` $\rightarrow$ `Tape` с очевидным смыслом. Каждая машина Тьюринга порождает конечные функции

```
NewControl :   Control × Alphabet → Control
NewSymbol :    Control × Alphabet → Alphabet
Displace :     Control × Alphabet → Displacement
```

Выбор этих трех функций (расширенных должным образом посредством значения по умолчанию `undef`) ведет нас к следующему высокоуровневому взгляду на машины Тьюринга:

```
par
   CurrentControl := NewControl(CurrentControl,Content(Head))
   Content(Head) := NewSymbol(CurrentControl,Content(Head))
   Head := Head + Displace(CurrentControl,Content(Head))
```

7.2.3. Нормальные алгоритмы Маркова

Нормальный алгоритм A оперирует со строками над данным алфавитом Σ. Он определяется последовательностью продукций

$$x_1 \rightarrow y_1, \ldots, x_n \rightarrow y_n,$$

где каждое x_i и каждое y_i являются Σ-строками. Некоторые продукции могут быть помечены как финальные. Состояниями A являются Σ-строки. Для того чтобы исполнить один шаг алгоритма A в состоянии Z, нужно сделать следующее. Проверить, является ли хотя бы один x_i непрерывным сегментом Z. Если нет, то остановка. Если да, найти наименьшее такое i и найти первое вхождение x_i в Z. Пусть $\mathtt{Pre}(x_i, Z)$ — часть Z перед первым вхождением x_i и $\mathtt{Post}(x_i, Z)$ — часть Z после первого вхождения x_i такое, что

$$Z = \mathtt{Pre}(x_i, Z) * x_i * \mathtt{Post}(x_i, Z),$$

где $*$ — операция конкатенации на строках. Заменим Z на новое состояние

$$\mathtt{Pre}(x_i, Z) * y_i * \mathtt{Post}(x_i, Z).$$

Если i-я продукция финальная, то остановка.

Нормальные алгоритмы не удовлетворяют принципу Ганди Локальной причинности. “Процесс решения о применимости некоторой подстановки к данному слову является существенно глобальным” [15]. Пусть

$\texttt{Occurs}(x, Z)$ — булевский тест, проверяющий, входит ли x в Z как непрерывный сегмент. Марков трактовал этот тест и функции $\texttt{Pre}$, $\texttt{Post}$, $*$ как уже данные. Мы будем поступать так же. Введем универсум $\texttt{String}$ всех строк над алфавитом Σ и положим, что $\texttt{Pre}(x, Z) = \texttt{Post}(x, Z) = \texttt{undef}$, если $\texttt{Occurs}(x, Z) = \texttt{false}$.

Легко понять, как запрограммировать любой нормальный алгоритм над Σ. Например, нормальный алгоритм

$$x_1 \to y_1, x_2 \to y_2, x_3 \to y_3,$$

в котором нет финальных продукций, может быть запрограммирован таким образом:

$$\begin{array}{lll}
\texttt{if} & \texttt{Occurs}(x_1, Z) \ \texttt{then} & Z := \texttt{Pre}(x_1, Z) * y_1 * \texttt{Post}(x_1, Z) \\
\texttt{else if} & \texttt{Occurs}(x_2, Z) \ \texttt{then} & Z := \texttt{Pre}(x_2, Z) * y_2 * \texttt{Post}(x_2, Z) \\
\texttt{else if} & \texttt{Occurs}(x_3, Z) \ \texttt{then} & Z := \texttt{Pre}(x_3, Z) * y_3 * \texttt{Post}(x_3, Z)
\end{array}$$

7.3. Конструкция `let`

MAC-Правило R в словаре Υ может рассматриваться как алгоритм само по себе. Состояния R — произвольные Υ-структуры, и каждое состояние является начальным. $\tau_R(X) = X + \Delta(R, X)$. Соответственно, два правила в словаре Υ эквивалентны тогда и только тогда, когда они порождают одно и то же множество обновлений $\Delta(X)$ во всякой Υ-структуре X. Например,

$$\begin{array}{lll}
\texttt{if } \phi & & \texttt{par} \\
\quad \texttt{then } R_1 & \text{эквивалентно} & \quad \texttt{if } \phi \ \texttt{then } R_1 \\
\quad \texttt{else } R_2 & & \quad \texttt{if } \neg\phi \ \texttt{then } R_2
\end{array}$$

Заметим, что последнее правило более многословно, особенно в случае большого ϕ. Это приводит нас к разработке более лаконичной программы. `let`-Правило

$$\begin{array}{l}
\texttt{let } x = t \\
\quad R(x)
\end{array}$$

спасает от проблемы писать и вычислять терм t неоднократно. Здесь x — новый 0-местный функциональный символ, который не встречается ни в R, ни в t. Это один из примеров синтаксического сахара, используемого в программировании MAC. В упрощенной теории MAC, описанной выше,

$$\begin{array}{lll}
\texttt{let } x = t & \text{эквивалентно} & R(t) \\
\quad R(x) & &
\end{array}$$

где $R(t)$ — результат одновременной подстановки t для каждого вхождения x в $R(x)$.

7.4. Последовательная композиция и теорема о линейном ускорении

В обычных языках программирования последовательная композиция позволяет комбинировать куски различной детализации: деревья и леса, кустарники и травинки. Как результат, программы могут становиться трудными для понимания. Последовательная MAC-программа Π описывает только один вычислительный шаг. Этот необычный стиль программирования помогает придерживаться одного уровня абстракции[7].

В принципе, тем не менее, последовательная композиция может быть встроена в парадигму MAC. Один шаг правила

```
seq
    R1
    R2
```

состоит из двух последовательных подшагов. Второй подшаг может переписать некоторые изменения, сделанные на первом подшаге. Если случается конфликт на любом подшаге, шаг целиком прерывается.

Легко видеть, что программы в обогащенном языке удовлетворяют нашим трем постулатам и поэтому `seq` может быть удален. Фактически существует унифицированный способ удаления `seq`. Подобным образом можно доказать теорему о линейном ускорении для исходного языка MAC.

Теорема 7.1. *Для всякой программы* Π_1 *имеется программа* Π_2 *над тем же самым словарем такая, что* $\tau_{\Pi_2}(X) = \tau_{\Pi_1}(\tau_{\Pi_1}(X))$ *для любой* Υ*-структуры* X.

Существующие средства, исполняющие MAC, избегают конструкции `seq` частично для ясности и частично по техническим причинам. Большая часть средств реализует мощную параллельную конструкцию `do-for-all` [21, 22], которая делает реализацию `seq` дорогостоящей. Другие последовательноподобные конструкции, например `let`, были более популярны.

7.5. MAC и рекурсия

Общим вопросом является: могут ли MAC обрабатывать рекурсию? Мы оцениваем рекурсивные вычисления как являющиеся неявно мультиагентными и трактуем их соответственно [17].

[7] Этот необычный стиль может пригодиться для программирования высокораспределенных систем, где процесс вычислений некоторого агента может быть прерван в любой момент.

7.6. Несовместимые множества обновлений

Пусть Δ — несовместимое множество обновлений над структурой X. Мы полагаем, что $X + \Delta = X$, так что эффект Δ совпадает с пустым множеством обновлений. Имеются содержательные альтернативы этому предположению. Несовместность можно выявлять и представлять различными способами в зависимости от обстоятельств [21]. На практике простое предположение, упомянутое выше, выглядит работоспособным. Если в этом есть необходимость, методы обнаружения несовместимости могут быть запрограммированы. В любом случае несовместимые множества обновлений не возникают в доказательстве основной теоремы, и поэтому эта проблема не относится к данной работе.

7.7. Типизированные машины абстрактных состояний

Структуры первого порядка являются многогранными. Наша версия понятия структуры поддерживает неформальную типизацию. Однако имеются прагматические причины ввести явную типизацию. Большая часть языков программирования следуют по этому пути. Проблема типизированных МАС исследовалась и исследуется. Результаты работ представлены на сайте [1]. Вы найдете там средства, исполняющие МАС (например, AsmGofer или ASM Workbench), которые используют типизированные версии языка программирования МАС.

8. АЛГОРИТМЫ, ВЗАИМОДЕЙСТВУЮЩИЕ СО СВОИМ ОКРУЖЕНИЕМ

До сих пор мы рассматривали неинтерактивные алгоритмы. Сейчас мы включим активные окружения в эту картину. Определение алгоритма, как нечто, что удовлетворяет трем постулатам (определение 6.1), не изменяется. Определение эквивалентности алгоритмов (определение 3.2) также не меняется. А что меняется, так это определение вычислительных последовательностей. Останется верным следствие 3.3 (эквивалентные алгоритмы имеют одни и те же вычислительные последовательности). Останется верной Основная теорема.

8.1. Активное окружение

8.1.1. Пример

Следующая интерактивная версия алгоритма Евклида многократно вычисляет наибольший общий делитель:

```
par
   if Mode = Initial then
      a := Input1, b := Input2, Mode := Compute
   if Mode = Compute then
      if b = 0 then d := a, Mode := Wait
      else if b = 1 then d := 1, Mode := Wait
      else a := b, b := a mod b
```

В состоянии с `Mode = Wait` алгоритм ожидает, когда окружение вмешается для того, чтобы изменить `Mode` на `Initial` и обновить `Input1` и `Input2`. Предполагается, что окружение удовлетворяет следующим ограничениям: оно вмешивается, только когда `Mode = Wait`; единственными функциями, на которые оно воздействует, являются `Mode`, `Input1` и `Input2`; оно может изменить `Mode` только на `Initial`; `Input1` и `Input2` должны быть натуральными числами. Эти ограничения позволяют окружению быть недетерминированным. Возможно, например, что третья пара на входе равна первой паре, но четвертая отличается от второй.

8.1.2. Агенты

Можно рассматривать вычисления интерактивной программы как игру с участием двух игроков, в которой один игрок — агент, исполняющий нашу программу, назовем его α, и другой игрок — активное окружение. В реальности в нее могли бы быть вовлечены множество агентов. Отношения между агентами могут включать как взаимодействие, так и соперничество за ресурсы. Кроме того, могут создаваться новые агенты и удаляться старые. В других случаях мы имеем дело с множественными агентами явно [21]. В последовательной парадигме агенты являются неявными. Окружение отвечает за действия всех внешних агентов (т.е. всех агентов, отличных от нашего агента α) так же, как и за некоторые глобальные ограничения.

8.2. Интерактивные вычислительные последовательности

Во время исполнения интерактивный алгоритм может получать и выдавать данные. Мы можем абстрагироваться от проблемы вывода данных. С точки зрения нашего алгоритма, вывод чудесным образом удаляется. Для того чтобы учесть проблему ввода данных, мы модифицируем определение вычислительной последовательности, данное в разд. 1. Вычислительная последовательность неинтерактивного алгоритма может рассматриваться как последовательность

$$\mu_1, \mu_2, \mu_3, \ldots$$

нехолостых переходов (или нехолостых шагов) алгоритма. Переходы интерактивного алгоритма A могут быть разбросаны между переходами окружения. Например, A может иметь последовательность

$$\mu_1, \nu_1, \mu_2, \nu_2, \mu_3, \nu_3, \ldots,$$

в которой A делает переходы μ и окружение делает переходы ν, так что мы имеем последовательность

$$X_0, X_1, X_1', X_2, X_2', X_3, X_3', \ldots$$

состояний, где X_0 — начальное состояние, $X_1 = \tau_A(X_0)$, X_1' получается из X_1 посредством некоторого действия окружения, $X_2 = \tau_A(X_1')$, X_2' получается из X_2 некоторого действия окружения и т.д.

Может показаться странным, что окружение должно действовать только между шагами алгоритма. В примере 8.1.1 окружение может приготовить новые `Input1` и `Input2` в то время, когда алгоритм работает над старыми `Input1` и `Input2`. Наше определение вычислительных последовательностей отражает точку зрения исполнителя данного алгоритма. Он не увидит новые входные данные, пока не закончит работу со старыми. С его точки зрения, новые входные данные появляются после того, как он закончит со старыми. Различные агенты могут видеть одни и те же вычисления существенно различными.

Можно привести доводы в пользу того, что вычислительная последовательность не может начинаться и заканчиваться с перехода окружения, что за каждым переходом окружения должен следовать переход алгоритма. Последнее является разумным требованием, но эти детали несущественны для наших целей в данной работе.

Легко видеть, что Основная теорема (теорема 6.13) остается верной. Нет необходимости пересматривать ни один из постулатов.

8.3. Новые элементы

В соответствии с дискуссией о неизменяемых базовых множествах в разд. 4 новые элементы импортируются окружением, и для этой цели используется специальная внешняя функция. Иногда бывает удобно скрыть эту внешнюю функцию. Для этого используются `create` (прежде `import`) правила [19]:

`create x`
$R(x)$

Обычно, правило `create` используется для расширения некоторого отдельного универсума U, которое дает начало сокращению `extend U`. Например, рассмотрим машину Тьюринга с конечной лентой, на которой

`Last` — крайне правая ячейка. Команда может включать расширение ленты:

```
extend Tape with x
   par
      Successor(Last) := x
      Predecessor(x) := Last
      Last := x
      ...
```

Вместо импортирования новых элементов из резерва (или из внешнего мира, если это имеет значение) удобно иметь дело с виртуальным миром, который содержит все, что может понадобиться для вычислений. Вы никогда не создаете новые элементы, вы скорее их находите или активизируете [3].

8.4. Дискуссия

В приложениях окружение обычно удовлетворяет строгим условиям. Определение словаря позволяет маркировать функциональные символы как *статические*. Эти символы не изменяются во время вычислений. В примере, упомянутом выше, было бы естественным объявить статическими 0-местные функциональные символы $0, 1$, `Initial`, `Compute`, `Wait` и бинарный функциональный символ `mod`. Другим типичным ограничением является то, что окружение удовлетворяет некоторой типовой дисциплине.

Базовые динамические функции (те, которые могут меняться во время вычислений) разбиты на:

- *внутренние*, те, чьи значения могут быть изменены только алгоритмом;
- *внешние*, те, чьи значения могут быть изменены только окружением;
- *разделяемые*, те, чьи значения могут быть изменены как окружением, так и алгоритмом.

(В некоторых приложениях удобно называть внутренние функции *управляемыми*, а внешние функции — *наблюдаемыми*[8] [9].) В примере, упомянутом выше, a, b, d — внутренние, `Input1` и `Input2` — внешние и `Mode` — разделяемая. Та же самая терминология применима к элементам памяти.

Исполнителю нашего алгоритма может понадобиться обратиться к окружению в середине выполнения шага. Например, после вычисления различных условий исполнитель может обнаружить, что ему необходимо окружение для того, чтобы создать новые элементы и/или что-то

[8] В оригинале *controlled* и *monitored* соответственно. — *Прим. перев.*

выбрать. Конечно, мы можем считать, что окружение сделало всю необходимую работу перед тем, как алгоритм начал исполнение шага. Но усовершенствование нашего определения интерактивных вычислительных последовательностей могло бы быть полезным. Эта проблема будет обсуждаться в будущей статье.

9. НЕДЕТЕРМИНИРОВАННЫЕ ПОСЛЕДОВАТЕЛЬНЫЕ АЛГОРИТМЫ

9.1. Каким образом алгоритмы могут быть недетерминированными?

Недетерминированные алгоритмы полезны, например, как высокоуровневые описания (спецификации) сложных детерминированных алгоритмов. Но алгоритму полагается быть точным описанием процесса исполнения. Каким образом он может быть недетерминированным? Позвольте нам проанализировать это кажущееся противоречие.

Вообразим, что вы исполняете недетерминированный алгоритм A. В некотором состоянии у вас есть несколько вариантов ваших действий и вы должны выбрать одну из доступных альтернатив. Программа алгоритма A сообщает вам, что вы должны выбрать, но не дает никаких предписаний, как это сделать. Что можете сделать вы? Можете поимпровизировать и, например, бросить жребий, для того чтобы указать вариант. Можете написать вспомогательную программу, которая сделает выбор за вас. Можете попросить кого-нибудь сделать этот выбор. Что бы вы ни делали, вы привносите нечто внешнее в алгоритм. Другими словами, это нечто — активное окружение, которое делает выбор. Наблюдаемые функции могут использоваться для того, чтобы указать выборы, сделанные активным окружением. В этом смысле недетерминизм уже доступен. Недетерминированные алгоритмы — это специальные интерактивные алгоритмы.

9.2. Недетерминизм с ограниченным выбором

Бывает удобно полагать, что варианты выбираются самим алгоритмом, а не окружением. В случае последовательных алгоритмов, где предполагается, что одношаговое преобразование работает ограниченное время, естественно требовать, что недетерминизм является ограниченным, так что на каждом шаге выбор ограничен предопределенным количеством альтернатив. Недетерминированные машины Тьюринга являются такими недетерминированными алгоритмами с ограниченным выбором.

Слегка модифицируем постулаты Последовательного времени и Аб-

страктного состояния, для того чтобы получить их недетерминированные версии. Теперь τ_A — бинарное отношение над $\mathcal{S}(A)$ и последние две части недетерминированного постулата Абстрактного состояния следующие:

— если $(X, X') \in \tau_A$, то базовое множество X' такое же, как у X;

— $\mathcal{S}(A)$ и $\mathcal{I}(A)$ замкнуты относительно изоморфизмов. Далее, пусть ζ — изоморфизм из состояния X на состояние Y. Для всякого состояния X' с $(X, X') \in \tau_A$ имеется состояние Y' с $(Y, Y') \in \tau_A$ такое, что ζ — изоморфизм из X' на Y'.

Недетерминированный алгоритм с ограниченным выбором — это объект A, который удовлетворяет недетерминированным постулатам Последовательного времени и Абстрактного состояния и прежнему постулату Локальности решения. Проверьте, что для всякого такого алгоритма A имеется натуральное число m такое, что для всякого $X \in \mathcal{S}(A)$ множество $\{Y : (X, Y) \in \tau_A\}$ содержит не более m элементов.

Расширим язык программирования детерминированных последовательных MAC конструкцией `choose-among`:

```
choose among
    R_1
    ...
    R_k
```

Назовем недетерминированные алгоритмы с ограниченным выбором A и B *эквивалентными*, если $\mathcal{S}(A) = \mathcal{S}(B)$, $\mathcal{I}(A) = \mathcal{I}(B)$ и $\tau_A = \tau_B$. Легко увидеть, что Основная теорема остается верной в случае недетерминированных алгоритмов с ограниченным выбором.

Более мощная конструкция выбора, которая может требовать неограниченного количества вариантов, дана в [21].

СПИСОК ЛИТЕРАТУРЫ

1. **ASM** Michigan Webpage. http://www.eecs.umich.edu/gasm/, maintained by J. K. Huggins.
2. **Григорьев Д.** Алгоритмы Колмогорова сильнее машин Тьюринга // Исследования по конструктивной математике и логике. Т. VII / Ред. Ю. Матиясевич и А. Слисенко / Записки научного семинара Ленинградского отделения Математического института им. Стеклова. Т. 60. — Л.: Наука; Ленингр. отд-ние, 1976. — С. 29–37.
3. **Blass A., Gurevich Y., Shelah S.** Choiceless polynomial time // Annals of Pure and Applied Logic. — 1999. — Vol. 100, N 1-3. — P. 141–187.
4. **Blass A., Gurevich Y.** The linear time hierarchy theorem for RAMs and abstract state machines // Springer J. of Universal Comput. Sci. — 1997. — Vol. 3, N 4. — P. 247–278.
5. **Blum L., Shub M., Smale S.** On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines // Bull. of American Mathematical Society. — 1989. — Vol. 21, N 1. — P. 1–46.

6. **Börger E., Grädel E., Gurevich Y.** Classical Decision Problem. — Berlin, Heidelberg, Germany: Springer-Verlag, 1996.
7. **Börger E., Huggins J. K.** Abstract state machines 1988–1998: Commented ASM bibliography // Bull. of European Association for Theor. Comput. Sci. — 1998. — N 64. — P. 105–128.
8. **Börger E.** Why use evolving algebras for hardware and software engineering? // Proc. of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics / Ed. by M. Bartosek, J. Staudek, and J. Wiederman. — Berlin, Heidelberg, Germany: Springer-Verlag, 1995. — P. 236–271. — (Lect. Notes Comput. Sci.; Vol. 1012).
9. **Börger E.** High level system design and analysis using abstract state machines // Proc. of FM-Trends'98, Current Trends in Applied Formal Methods / Ed. by D. Hutter, W. Stephan, P. Traverso, M. Ullmann. — Berlin, Heidelberg, Germany: Springer-Verlag, 1999. — P. 1–43. — (Lect. Notes Comput. Sci.; Vol. 1641).
10. **Chandy K. M., Misra J.** Parallel Program Design. — Reading, MA: Addison-Wesley, 1988.
11. **Church A.** An unsolvable problem of elementary number theory // American J. of Math. — 1936. — Vol. 58. — P. 345–363.
12. **Cook S. A., Reckhow R. A.** Time bounded random access machines // J. of Comput. and Systems Sci. — 1973. — Vol. 7, N 4. — P. 354–375.
13. **Davis M.** The Undecidable. — NY: Raven Press, 1965.
14. **Колмогоров А. Н., Успенский В. А.** К определению понятия алгоритма // Успехи математических наук. — 1958. — Т. 13, № 4. — С. 3–28.
15. **Gandy R.** Church's thesis and principles for mechanisms // The Kleene Symp / Ed. by J. Barwise, H. J. Keisler, K. Kunen. — Amsterdam: North-Holland, 1980. — P. 123–148.
16. **Gries D.** The maximum-segment-sum problem // Formal Development of Programs and Proofs / Ed. by E. W. Dijkstra — Reading, MA: Addison-Wesley, 1990. — P. 33–36.
17. **Gurevich Y., Spielmann M.** Recursive abstract state machines // Springer J. of Universal Comput. Sci. — 1997. — Vol. 3, N 4. — P. 233–246.
18. **Gurevich Y.** A new thesis // American Mathematical Society Abstracts. — 1985. — Vol. 6, N 4, Iss. 39. — P. 317, abstract 85T-68-203.
19. **Gurevich Y.** Evolving algebras: An attempt to discover semantics // Current Trends in Theor. Comput. Sci. / Ed. by G. Rozenberg, A. Salomaa. — World Scientific, 1993. — P. 266–292.
20. **Gurevich Y.** Kolmogorov machines and related issues // Ibid. — P. 225–234.
21. **Gurevich Y.** Evolving algebra 1993: Lipari guide // Specification and Validation Methods / Ed. by E. Börger — Oxford: Oxford University Press, 1995. — P. 9–36.
22. **Gurevich Y.** May 1997 draft of the ASM guide. — 1997. — (Tech. Rep. / EECS Department, College of Engineering, Univ. of Michigan; CSE-TR-336-97).
23. **Gurevich Y.** The sequential ASM thesis // Bull. of European Association for Theor. Comput. Sci.. — 1999. — N 67. — P. 93–124.
24. **Марков А. А.** Теория алгоритмов. — М.: Наука, 1954. — (Тр. Математического института им. Стеклова; Т. 42).
25. **Knuth D. E.** Fundamental Algorithms. Vol. 1 of The Art of Computer Programming. — Reading, MA: Addison-Wesley, 1968.
26. **Savage J. E.** The Complexity of Computing. — Malabar, FL: Robert E. Krieger, 1987.
27. **Savage J. E.** Models of Computation: Exploring the Power of Computing. — Boston, MA: Addison Wesley Longman, 1998.
28. **Schönhage A.** Storage modification machines // SIAM J. on Comput. — 1980. — Vol. 9. — P. 490–508.

29. **Tarski A.** The concept of truth in the languages of deductive sciences // Logic, Semantics, Metamathematics: Papers from 1923 to 1938 / Ed. by J. H. Woodger. — Oxford: Clarendon Press, 1956. — P. 152–278.
30.* **Blass A., Gurevich Y.** Abstract State Machines Capture Parallel Algorithms // ACM Trans. on Computation Logic. — 2003. (в печати).
31.* **Gurevich Y.** Sequential Abstract State Machines Capture Sequential Algorithms // ACM Trans. on Computational Logic. — 2000. — Vol. 1, N 1. — P. 77–111.
32. **Turing A.** On computable numbers, with an application to the Entscheidungsproblem // Proc. of London Mathematical Society. — 1936. — Vol. 2, N 42. — P. 230–236. Исправления: *Ibid.* — N 43. — P. 544–546.
33. **Uspensky V. A.** Kolmogorov and mathematical logic // J. of Symbolic Logic. — 1992. — Vol. 57, N 2. — P. 385–412.

Приложение

КОНЕЧНАЯ ЛОКАЛЬНОСТЬ РЕШЕНИЯ

Предполагая выполненными постулаты Последовательного времени и Абстрактного состояния, мы выведем постулат Локальности решения из гипотезы, которая является, по-видимому, более слабым предположением. Для этого мы не требуем в Приложении, чтобы алгоритм удовлетворял постулату Локальности решения. Алгоритм определяется как любой объект, удовлетворяющий постулатам Последовательного времени и Абстрактного состояния. Логически это Приложение принадлежит концу разд. 5, но мы его отложили, для того чтобы избежать прерывания основного обсуждения.

Определение A.1. *Алгоритм A в словаре Υ является конечно-локальным, если имеется отображение $\mathcal{T}$, которое присваивает конечное множество $\mathcal{T}(X)$ Υ-термов каждому состоянию X алгоритма A таким образом, что выполнены следующие два требования:*

(1) $\Delta(A, Y) = \Delta(A, X)$ для всякого состояния Y алгоритма A, который совпадает с X над $\mathcal{T}(X)$;

(2) если X, Y — два состояния алгоритма A, то либо $\mathcal{T}(X) = \mathcal{T}(Y)$, либо имеются термы $t_1, t_2 \in \mathcal{T}(X) \cap \mathcal{T}(Y)$, чьи значения равны в одном из состояний и различны в другом.

Интуитивно, первое требование соответствует тому, что на каждом шаге A проверяет только конечную часть текущего состояния X, которая определяется термами из $\mathcal{T}(X)$. Заметим, что множество $\mathcal{T}(X)$ может изменяться с изменением состояния X.

Второе требование отражает детерминизм алгоритма A. Андреас Бласс предложил следующий сценарий. Состояние — это база данных. На каждом шаге агент, исполняющий A, делает запрос к базе данных и вычисляет подходящие обновления. Обновления могут передаваться в

базу данных, например, в форме правил обновления. Но что агент может узнать? С тех пор как (ввиду постулата Абстрактного состояния) внутренние представления элементов состояния не важны, агент может только узнать, что такие-то и такие-то термы имеют одно и то же значение, а такие-то и такие-то термы имеют разные значения. Вычисления (одношаговые) агента на состояниях X и Y могут разветвиться, если агент обнаружит, что некоторые термы t_1, t_2 имеют одинаковые значения в одном из состояний и различные значения в другом. В противном случае вычисления идентичны, и поэтому $\mathcal{T}(X) = \mathcal{T}(Y)$.

Назовем алгоритм A *локальным*, если он удовлетворяет постулату Локальности решения. Другими словами, A — локальный тогда и только тогда, когда он конечно-локальный с конечно-локальным свидетелем $\mathcal{T}$ таким, что $\mathcal{T}(X)$, не зависит от X.

Что произойдет, если конечно-локальный алгоритм A с конечно–локальным свидетелем $\mathcal{T}$ применяется к Υ-структуре Y, которая не является состоянием A? Поскольку A и Y имеют один и тот же словарь, запросы исполняющего агента могут иметь смысл в Y. Агент может узнать, что такие-то и такие-то термы имеют одно и то же значение в Y и что такие-то и такие-то термы имеют различные значения в Y. Появляются две возможности.

- Существует состояние X такое, что Y совпадает с X над $\mathcal{T}(X)$. В этом случае Y можно было бы назвать *псевдосостоянием* алгоритма A (по отношению к $\mathcal{T}$).
- Не существует состояния X такого, что Y совпадает с X над $\mathcal{T}(X)$. В этом случае Y можно было бы назвать *ошибочным состоянием*[9] алгоритма A (по отношению к $\mathcal{T}$).

Для выразительности состояния алгоритма A можно назвать *законными состояниями*. Нам интересны алгоритмы, способные различать законные состояния и ошибочные состояния в следующем смысле.

Определение A.2. *Алгоритм A над словарем Υ является различающим, если имеется отображение $\mathcal{T}$, которое присваивает конечное множество $\mathcal{T}(X)$ Υ-термов каждой Υ-структуре X таким образом, что выполнены следующие требования:*

[9] Ошибочное состояние — это не состояние. Такая терминология не является редкостью. — *Прим. перев.*: Далее в этом примечании автор приводит некоторый фразеологический оборот, дословно означающий копченую селедку, а в переносном смысле — отвлекающий маневр. — A red herring is often not a herring and not necessarily red either.

— *$\mathcal{T}$ — конечно-локальный свидетель*[10] *для A;*
— *если Y — ошибочное состояние алгоритма A по отношению к $\mathcal{T}$, то не существует законного состояния алгоритма A, совпадающего с Y над $\mathcal{T}(Y)$.*

Если A — локальный алгоритм с локальным свидетелем $\mathcal{T}$, то A является различающим с $\mathcal{T}(X) \rightleftharpoons T$.

Пример А.3. *Предположим, что A — алгоритм над словарем Υ, чья нелогическая часть состоит из 0-местного функционального символа 0, унарного функционального символа f и 0-местного предикатного символа* `parity`. *Υ-структура X — законное состояние A, если множество $S(X)$ значений термов $0, f(0), f(f(0)), \ldots$ — конечно в X. Состояние X алгоритма A — начальное, если* `parity = false` *в X. A выполняет только один шаг. Если мощность $S(X)$ — четна,* `parity` *устанавливается в* `true`*, иначе никаких изменений не происходит.*

Легко видеть, что A — конечно-локальный. Однако A не является различающим. Рассмотрим Υ-структуру Y, где термы 0, $f(0)$, $f(f(0)), \ldots$ имеют различные значения. Y — ошибочное состояние, но для всякого множества T Υ-термов имеется достаточно большое законное состояние алгоритма A, которое совпадает с Y над T.

Теорема А.4. *Всякий различающий алгоритм является локальным.*

Доказательство. Пусть A — различающий алгоритм. Получим локального свидетеля T для A. Воспользуемся

— теоремой компактности для логики первого порядка, которая может быть найдена в стандартных учебниках по математической логике, и
— утверждением о компактности канторова пространства, доказываемым в стандартных учебниках по топологии.

Напомним, что канторово пространство состоит из бесконечных последовательностей $\xi = \langle \xi_i : i \in N \rangle$, где N — множество $\{0, 1, \ldots\}$ натуральных чисел и каждое $\xi_i \in \{0, 1\}$. Всякая функция f из конечного множества натуральных чисел в $\{0, 1\}$ дает базовое открытое множество

$$O(f) \rightleftharpoons \{\xi : \xi_i = f(i) \text{ для всех } i \in \mathrm{Domain}(f)\}$$

канторова пространства. Произвольное открытое множество канторова пространства является объединением набора базовых открытых множеств $O(f)$.

Пусть Υ — словарь алгоритма A и $\mathcal{T}$ — различающий свидетель для A. В этом доказательстве термы и структуры определены над словарем Υ,

[10] Или сужение $\mathcal{T}$ на множество законных состояний является конечно-локальным свидетелем. Мы не должны требовать в определении А.1, чтобы область $\mathcal{T}$ была ограничена на состояния A.

состояния — состояния алгоритма A, псевдосостояния — псевдосостояния A по отношению $\mathcal{T}$, ошибочные состояния — ошибочные состояния A по отношению $\mathcal{T}$.

Лемма А.5. *Предположим, что X, Y — законные состояния и Z — псевдосостояние. Если Z совпадает с X над $\mathcal{T}(X)$ и с Y над $\mathcal{T}(Y)$, то $\mathcal{T}(X) = \mathcal{T}(Y)$.*

Доказательство. Все три структуры совпадают над $\mathcal{T}(X) \cap \mathcal{T}(Y)$. Из второго условия определения А.1 следует, что $\mathcal{T}(X) = \mathcal{T}(Y)$. ■

Определение А.2 не накладывает ограничений на множества $\mathcal{T}(Y)$, где Y является псевдосостоянием. Принимая во внимание лемму А.5, мы можем предполагать следующее: если псевдосостояние Y совпадает с законным состоянием X над $\mathcal{T}(X)$, то $\mathcal{T}(Y) = \mathcal{T}(X)$.

Следствие А.6. *Если законное состояние X совпадает со структурой Y над $\mathcal{T}(Y)$, то $\mathcal{T}(X) = \mathcal{T}(Y)$.*

Доказательство. Из второго условия определения А.2, Y не может быть ошибочным состоянием. Рассмотрим два оставшихся случая.

Случай 1: Y — законное состояние. Используем второе условие определения А.1.

Случай 2: Y — псевдосостояние. Тогда существует законное состояние Z такое, что Y совпадает с Z над $\mathcal{T}(Z)$, и поэтому $\mathcal{T}(Y) = \mathcal{T}(Z)$. Следовательно, X совпадает с Z над $\mathcal{T}(Z)$. Тогда, как и в случае 1, $\mathcal{T}(X) = \mathcal{T}(Z)$. Следовательно, $\mathcal{T}(X) = \mathcal{T}(Y)$. ■

Назовем терм *эквациональным*, если он имеет вид $t_1 = t_2$. Список всех эквациональных термов в некотором порядке:

$$e_1, e_2, e_3, \ldots \ .$$

Каждая структура X дает бинарную последовательность χ^X, где $\chi_i^X = 1$, если e_i верно в X, и $\chi_i^X = 0$, если e_i не верно в X.

Пусть $O(X)$ — базовое открытое множество канторова пространства, которое состоит из последовательностей ξ с $\xi_i = \chi_i^X$ для всякого равенства e_i такого, что обе части e_i принадлежат $\mathcal{T}(X)$.

Лемма А.7. *Пусть ξ — бинарная последовательность, которая не принадлежит никакому $O(X)$. Существует открытое множество $O(\xi)$, которое содержит ξ и не пересекается ни с каким $O(X)$.*

Доказательство. Рассмотрим аксиоматическую систему первого порядка в словаре Υ, которая содержит обычные аксиомы равенства и включает бесконечное множество

$$D \rightleftharpoons \{\phi_i : i = 1, 2, 3, \ldots\},$$

где ϕ_i является e_i, если $\xi_i = 1$, и ϕ_i является $\neg e_i$, если $\xi_i = 0$.

Если всякое конечное подмножество D выполнимо, то по теореме о компактности для логики первого порядка D выполнимо, и поэтому оно имеет модель X. Но тогда $\xi \in O(X)$, что противоречит нашему предположению относительно ξ.

Следовательно имеется конечный невыполнимый фрагмент D_0 подмножества D. Требуемое $O(\xi)$ состоит из всех последовательностей ξ' таких, что $\xi'_i = \xi_i$ для всех i с $\phi_i \in D_0$. ■

Открытые множества $O(X)$ и $O(\xi)$ покрывают канторово пространство. Поскольку канторово пространство компактно, оно покрывается конечным набором этих открытых множеств. Поэтому имеется конечный набор $\{X_1, \ldots, X_k\}$ состояний таких, что

$$\{\chi^X : X \text{ — состояние}\} \subseteq O(X_1) \cup \cdots \cup O(X_k).$$

Множество $T = \mathcal{T}(X_1) \cup \cdots \cup \mathcal{T}(X_k)$ является свидетелем того, что A локальный. В самом деле, пусть состояния Y и Z совпадают над T. Существует $i \in \{1, \ldots, k\}$ такое, что $Y \in O(X_i)$, так что Y совпадает с X_i над $\mathcal{T}(X_i)$. Из следствия A.6 $\mathcal{T}(Y) = \mathcal{T}(X_i)$, так что $\mathcal{T}(Y) \subseteq T$. Следовательно Z совпадает с Y над $\mathcal{T}(Y)$. Ввиду первого условия определения A.1 $\Delta(A, Y) = \Delta(A, Z)$. Это завершает доказательство теоремы. ■

Пусть A — конечно-локальный алгоритм с конечно-локальным свидетелем $\mathcal{T}$ и пусть Υ — словарь A. Как и в доказательстве теоремы A.4, термы и структуры определены над словарем Υ, состояния — состояния A и псевдосостояния (ошибочные состояния) — псевдосостояния (ошибочные состояния) A по отношению к $\mathcal{T}$.

Назовем две структуры *подобными*, если не существует эквационального терма, различающего эти состояния, т.е. если некоторое равенство выполнено в одной из структур, то оно выполнено и в другой. Для любой структуры X пусть $[X]$ — набор всех структур Y, подобных X (*класс подобности* X).

Лемма A.8. *Всякая структура, подобная некоторому ошибочному состоянию, является ошибочным состоянием.*

Доказательство. Предположим, что структура Z подобна ошибочному состоянию Y. Если Z — законное состояние или псевдосостояние, то имеется законное состояние X такое, что Z совпадает с X над $\mathcal{T}(X)$. Поскольку Y подобно Z, Y также совпадает с X над $\mathcal{T}(X)$. Таким образом, Y — псевдосостояние, что противоречит нашему предположению, что Y — ошибочное состояние. ■

Пусть отображение $X \mapsto \chi^X$ такое же, как в доказательстве теоремы А.4. Отображение $[X] \mapsto \chi^X$ из множества классов подобности в канторово пространство является однозначным (но не взаимно-однозначным). Используем его для того, чтобы определить топологическое пространство TS классов подобности структур. Открытые множества TS являются прообразами открытых множеств канторова пространства.

Теорема А.9. *Если имеется замкнутое множество F в TS, такое, что*

- *F содержит все классы подобности $[X]$, где X — законное состояние и*
- *F не содержит классы подобности $[Y]$, где Y — ошибочное состояние,*

то A — локальный.

Доказательство. Пусть Y — ошибочное состояние. Дополнение F в TS является открытым множеством, которое содержит $[Y]$, но не содержит никакого $[X]$, где $[X]$ — законное состояние. По определению TS имеется открытое подмножество канторова пространства, которое содержит χ^Y, но не содержит никакого χ^X, где X — законное состояние. Следовательно, имеется базовое открытое множество $O(f)$, которое содержит χ^Y, но не содержит никакого χ^X, где X — законное состояние. Определим $\mathcal{T}(Y)$ как множество всех подтермов эквациональных термов e_i с $i \in \mathrm{Domain}(f)$. Никакое законное состояние X алгоритма A не совпадает с Y над $\mathcal{T}(Y)$; в противном случае χ^X могло бы принадлежать $O(f)$, что невозможно.

Расширенное отображение $\mathcal{T}$ — различающий свидетель для A. По теореме А.4 алгоритм A — локальный. ■