УДК 510.2, 004.8

А. В. Самойлов

Композиция математических объектов в рамках абстрактной модели мышления

Рассматривается структура основных математических объектов, опирающаяся на абстрактную модель разумного мышления. Каковая, в свою очередь, выводится из основополагающего высказывания “я мыслю, следовательно, существую”. Показано, что данные аспекты математики взаимосвязаны и могут быть заданы только в рамках объединяющей теории.

Библиография: 3 названия.

Ключевые слова: абстрактная модель мышления, смешанные системы, математическая реальность, низшая математика, ФОРАОН, формальная семантика, темпоральная механика.

§ 1. Введение

Потребность в настоящей работе обусловлена комплексом проблем (открытых вопросов) современной науки. Эти проблемы относятся, с одной стороны, к теории искусственного интеллекта, а с другой – к основаниям математики. Опишем их вкратце.

В отношении ИИ, можно указать, что эта область знаний не имеет общепризнанного, твёрдого и законченного теоретического фундамента, имея в виду строгую математическую модель мышления: компоненты и структура мыслящей системы, её цели и алгоритмы их достижения. Человечество возлагает на ИИ большие надежды и направляет на его развитие значительные усилия. Поэтому, обнаружение математической модели мышления видится крайне актуальной задачей.

В области оснований математики, укажем проблему композиции. Математические конструкции могут быть весьма сложны, но всё сложное состоит из более простого, а в современной математике поиск наиболее простого обнаруживает сразу несколько объектов, среди которых, например, логические значения, мнимая единица, пустое множество, и т.д.. Разбиение подобных объектов на более простые не описано.

Но, незнание композиции *не эквивалентно* её отсутствию. Указанные объекты различаются в поведении, что говорит в пользу различий и в структуре. Исследовать данную область представляется совершенно необходимым: знание структуры объекта эквивалентно доказательству описывающих этот объект аксиом. Решение этой проблемы позволит научить ИИ *понимать* математику.

Работа по этим направлениям была начата в 2013 году. В ходе исследования было обнаружено, что две упомянутые проблемы могут быть решены *только совместно*. Как показало исследование, между математикой и мышлением

Посвящается моим родителям.

Помощь в исследовании оказал к.ф.-м.н. Ленюк С. В., доцент каф. выс. мат. МФТИ.

есть фундаментальная связь, ввиду которой предъявить композицию указанных математических объектов можно только в рамках модели мышления.

Изложение результатов проведённого исследования превышает по объёму журнальную статью, поэтому материал разделён на несколько частей, каждая из которых оформляется в виде отдельной статьи. Данная первая статья описывает решение проблемы композиции математических объектов.

§ 2. Философия

2.1. Принципы. Данная статья описывает и обсуждает абстрактную модель мышления (АММ)[1]. Слова "мышление" и "разум" полагаем синонимами.

Источником необходимости для АММ являются наши знания о мышлении: если модель верна, то мы можем безусловно ожидать от неё тех или иных черт. Мы будем неоднократно возвращаться к мышлению как к источнику необходимости; сейчас укажем на два аспекта, важных для всей статьи.

Первый аспект связан с тем, что мышление способно думать абсолютно обо всём, вне зависимости от существования предмета рассуждений. Данный тезис будем называть *принципом универсальности*.

Пример: вещественное число, отличное от нуля, либо положительно, либо отрицательно. Тогда пусть число, которое одновременно больше и меньше нуля, называется чудесным. Существуют такие числа или нет, принцип универсальности требует, чтобы сама возможность их формализации не выходила за рамки АММ, в противном случае эта модель не будет исчерпывающей.

Этот пример иллюстрирует действие принципа универсальности при обосновании необходимости. Если тем или иным образом обоснована необходимость некоторого правила, то принцип универсальности, в свою очередь, обосновывает необходимость нарушения этого правила.

Второй аспект связан с внутренним представлением знаний и независимостью мышления от внешних языков. Очевидно, что мышление ребёнка успешно функционирует даже до того, как он начинает понимать и применять свой первый язык, в противном случае он не мог бы этого сделать.

Чтобы подобное стало возможным для АММ, она должна обладать встроенным способом хранения знаний, который представляет структуру объектов напрямую, т.е. *явно* выражает их композицию: элементный и информационный состав. Данный тезис будем называть *принципом независимости*.

2.2. Смешанные системы. В самых разных системах, хранение и изменение каких-либо объектов суть две наиболее низкоуровневые, первичные, основополагающие функции. Для выполнения этих функций в системе должна присутствовать *память*, хранящая объекты, и *время*, их изменяющее. Тройку $\{m, t, x\}$ будем называть *смешанной системой*[2], где память обозначена как m, время как t, а объекты в памяти как x.

Функции хранения и изменения присутствуют повсеместно в природе. Так, одну из самых простых образуют кварки и глюоны: кварки (память) хранят цвет, который может быть изменён глюоном (время).

[1] Англ. "abstract model of mind (AMM)".
[2] Англ. "mixed system".

На другом краю шкалы сложности находятся мозг человека и знания. Нейроны выполняют роль памяти, а нервные импульсы роль времени. Отсюда происхождение термина "смешанная" система, которая состоит как из материальных (нейроны, импульсы), так и из абстрактных (знания) элементов.

Среди теоретических или искусственных систем, самые очевидными примерами являются машина Тьюринга и компьютер. В машине Тьюринга роли памяти, времени и объектов играют лента, головка и символы, соответственно. В ЭВМ данные в оперативной памяти изменяются процессором.

Альтернативный термин для смешанной системы – *вариальность*[3].

2.3. Математическая реальность. Поскольку разум человека является смешанной системой, АММ должна включать в себя представление памяти и времени. Задавшись вопросом о природе подобного представления, необходимо учесть, что оно должно отвечать принципам универсальности и независимости. Впрочем, такая модель обнаруживается достаточно легко: она прослеживается в истории человеческого интеллекта от доисторических времён до наших дней.

Одним из ранних свидетельств разума является наскальная живопись. Для неё нужна была поверхность (стена пещеры, возможно) и средство нанесения рисунка. Однажды появившись, средства письма остались с людьми навсегда, изменяясь технически, но не по сути.

Будем сокращённо называть поверхность для письма *доской*, а средство нанесения рисунка *маркером*. Смешанную систему, в которой доска является памятью, а маркер временем, будем называть *математической реальностью*. В данной модели, надписи на доске играют роль объектов смешанной системы.

В центре данной работы лежит идея использовать математическую реальность как представление памяти и времени в АММ и математике в целом. Плюсом этого подхода является то, что математика полагает средства письма самоочевидными.

2.4. Начальный постулат. Отправной точкой наших рассуждений является известное из философии высказывание "я мыслю, следовательно, существую" (начальный постулат, НП). Это утверждение Декарт выдвинул как первичную достоверность, истину, в которой невозможно усомниться.

Мы будем трактовать НП именно в таком ключе – высказывания, которое доказывает само себя, как выражение способности субъекта рассуждать и воспринимать внешний мир: исключительное явление в философии.

Данный тезис тесно связан с сопровождающей его обширной философской дискуссией, но у нас нет необходимости углубляться в неё, т.к. мы будем использовать крайне ограниченный набор математических следствий из данного философского вывода. Необходимо подчеркнуть, что эти следствия (3.1) соответствуют месту, где в данной работе стыкуются философия и математика.

Математическую теорию, вытекающую из начального постулата, будем называть *низшей математикой*[4]. Имея в виду столь надёжный фундамент, мы можем ожидать, что подобная теория будет в состоянии решить задачи, упомянутые во введении. Во-первых, формально описать композицию основных

[3]Англ. "variality".
[4]Англ. "lower mathematics".

математических объектов. И во-вторых, математически описать цели мышления и способы их достижения. Эти две части теории будем называть, соответственно, общей и специальной.

Данная статья описывает первую часть теории. Совокупность её определений будем называть "формальный аппарат общего назначения" (ФОРАОН)[5]. Определения даются при помощи оригинальной графической нотации и программного кода на языке C#, дополняющих друг друга. Для каждого вводимого объекта, мы указываем цепочку обоснования, т.е. путь вывода его необходимости из НП, а также элементный и информационный состав объекта.

Все необходимые для работы понятия вводятся явно и действительны в рамках низшей математики. Это также касается объектов, которые в рамках традиционной математики поясняются и/или задаются аксиоматически.

2.5. Формальная семантика. Традиционно, математика связывает понятие формальности с языками, ассоциируя с ними термины "символ", "алфавит", "слово". При этом, слова – только лишь *алиасы* сложной семантики объекта, а способ их сопоставления и образует, собственно, формальный язык.

Но, такой способ построения языка не обеспечивает передачу структуры.

Формальной семантикой будем называть способ представления смысла, явно (напрямую) выражающий структуру. Данная работа обнаруживает, что математическая реальность достаточно богата для этого.

2.6. Программная нотация. Программные определения, приведённые в статье, доступны на сайте по адресу https://russian.mixed.systems/FORAON. Объектам ФОРАОН даны названия в нотации `underscore_case`, а библиотечные классы и методы именуются в нотации `PascalCase`.

Мы используем язык программирования, чтобы *выразить структуру* излагаемой идеи, поскольку язык программирования на 100% однозначен, а также лаконичнее естественного языка. Код явно указывает композицию объекта, и других "неучтённых" элементов не подразумевается.

Необходимо подчеркнуть, что ключевые слова C#, такие как `public`, `void`, `class`, `static` и прочие черты языка присутствуют в коде только лишь для того, чтобы его можно было скомпилировать и запустить. Для целей статьи их нужно игнорировать. Нас будут интересовать следующие аспекты кода: имена, элементы сложных объектов и функции – то, что имеет прямые аналоги в математике. Остальные элементы языка программирования надо воспринимать так же, как мы воспринимаем клеточки в тетради.

§ 3. Структура АММ

3.1. Предел осознания. Приступим к разбору начального постулата. Мы будем анализировать как смысл, так и запись этой фразы.

Укажем, что НП высказан от первого лица, т.е. он имеет смысл для некоторого разумного субъекта. Чтобы отразить данный факт, в структуре АММ должна быть часть, соответствующая слову "я". Мы будем называть эту часть АММ термином *внимание*.

[5] Англ. "formal apparatus of overall notion" (FORAON).

Формальное определение внимания: `public class attention { }`. Для целей статьи достаточно задать его как класс без содержания – важно само наличие внимания в составе AMM; в специальной части теории это определение будет расширено. В формулах будем обозначать внимание символом $\underline{\varphi}$.

Слово “мыслю” обосновывает необходимость памяти и времени в составе AMM. Одновременно, из записи НП следует, что доска и маркер являются моделями этих понятий. Поскольку использование средств письма не вызывает у математиков затруднений, мы можем включить доску и маркер в AMM и применять их для обоснования необходимости.

Смешанная система (вариальность), входящая в состав мышления, моделируется классом `variality` (Прог. 1). Как видно из определения, членами этого класса являются доска и маркер (`board` и `marker`), подробнее описанные далее.

```
1  public partial class variality {
2
3      public static board memory;
4      public static marker time() => new(); }
```

Прог. 1: Вариальность

В свою очередь, `attention` и `variality` комбинируются в классе `sentience` (Прог. 2), отражающем структуру мышления в целом. Данный класс является корневым определением AMM. Таким образом, в состав AMM включены понятия, необходимость которых вытекает из “я мыслю”.

```
1  public partial class sentience {
2
3      public static attention personality;
4      public static variality mixed; }
```

Прог. 2: Интеллект

Обратимся к слову “существую”. Для человека, оно означает материальный характер мышления, присутствие разума в физическом мире. В математике, “существую” будем трактовать как указание на то, что входящие в AMM объекты образуют отдельную (в противоположность знаниям) категорию, аналогичную материальным элементам разума. Эти объекты будем называть *собственными*[6].

Исследуя сам себя, разум способен осознать собственные объекты: внимание, память и время, но и только. Обнаружить более детальную структуру (нейроны и импульсы) при таком исследовании нельзя. Мы будем называть границу самоочевидных знаний о мышлении *пределом осознания*.

3.2. Доска. Математика полагает использование поверхности для письма естественным, что даёт возможность использовать свойства доски при обосновании необходимости.

Для нас представляют интерес следующие аспекты поверхности (её размер не рассматривается):

[6] Англ. “own objects”.

- два измерения (оси) – первичной и вторичной ориентации;
- на каждой оси два направления – положительное и отрицательное;
- преобразование осей и направлений при помощи поворотов, имеющих направление.

В настоящей работе выбраны ориентации и направления, как показано на Рис. 1. Первичная и вторичная оси обозначены как $\boldsymbol{p}$ и $\boldsymbol{s}$, положительные и отрицательные направления как $+$ и $-$. По умолчанию принимаются: положительное направление и первичная ориентация.

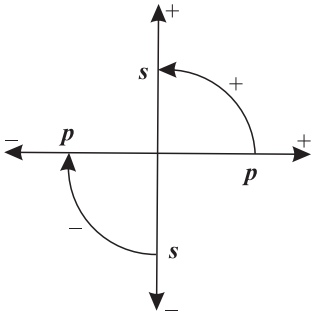


Рис. 1: Аспекты доски

Соответствующее программное определение приведено в Прог. 3.

```
1  public class board {
2      public enum drn { p = 0, n = 1, }
3      public enum orn { p = 0, s = 1, }
4      public delegate void rotate(object x, drn d); }
```

Прог. 3: Доска

В классе `board` направления и ориентации представлены соответственно как `drn` и `orn`, а поворот как делегат `rotate`. Направления и ориентации моделируются как перечисления (`enum`) со значениями $0, 1$. Это значит, что мы не видим в них каких-либо свойств, кроме отличия одного значения от другого.

В формулах будем обозначать доску символом $\mathscr{B}$.

3.3. Маркер. Время с необходимостью входит в понятие мышления. Оно делает возможными *действия*, в частности, изменения. Мы будем рассматривать время в отрыве от его физического аспекта, что делает возможным рассмотреть выполнение любой конечной или бесконечной цепочки действий.

Запись НП обосновывает использование маркера как модели времени.

Чтобы описать использование маркера, надо абстрагироваться от человека, стоящего с маркером у доски. В нашем рассмотрении есть только внимание, доска и маркер. Следовательно, в рамках АММ всё взаимодействие с доской должно выполняться маркером. Описание маркера начнём с базового сценария, в котором внимание инициирует запись какого-либо объекта на доске. Поведение маркера выглядит так:

- получить от внимания объект, подлежащий записи;
- записать этот объект на доске.

Мы исходим из того, что получение данных и последующая запись на доске отстоят во времени. Хотя мы не рассматриваем физический аспект времени, однако на концептуальном уровне будем выделять предыдущий и последующий моменты. В силу этого, маркер должен иметь встроенную память для хранения полученного объекта.

Всё вышесказанное записывается на C# как в Прог. 4.

```
1  public class marker {
2
3      public object argument;
4      public void function(object previous, object next) {
5
6          argument = read(previous);
7          write(next, argument); }}
```

Прог. 4: Маркер: базовый сценарий

Рассмотрим код подробнее. Класс `marker` содержит два члена: поле `argument`, представляющее собой память маркера, и метод `function`, описывающий поведение маркера.

Маркер взаимодействует с двумя объектами, которые будем называть *коммутаторами*: предыдущий (`previous`) и следующий (`next`). В базовом сценарии предыдущим коммутатором является внимание, следующим – доска (вместе с тем, принцип универсальности требует, чтобы маркер не ограничивал диапазон возможных в качестве коммутаторов объектов).

Метод `function` состоит из двух частей. Первая (с. 6) читает информацию из предыдущего коммутатора и копирует её в память маркера. Вторая – записывает её в следующий коммутатор (с. 7).

Важный аспект маркера, вытекающий из базового сценария: как и всякое средство записи, маркер должен предоставить гарантию, что предыдущий коммутатор не будет изменён: действительно, процесс записи на доске не должен изменять содержание внимания (чтение не должно изменять объект).

Для этого, необходимо обработать случай, когда один и тот же объект передан маркеру в качестве как предыдущего, так и следующего коммутатора. В этом случае запись нужно отменить[7], как в Прог. 5, с. 4. На языке C# это выражается через аппарат исключения.

```
1      public void function(object previous, object next) {
2
3          argument = read(previous);
4          if (previous == next) throw new Exception();
5          write(next, argument); }
```

Прог. 5: Маркер: защита предыдущего коммутатора

На этом базовый сценарий завершён.

Чтобы произвести чтение, достаточно лишь выполнить маркер, передав доску в качестве предыдущего, а внимание в качестве следующего коммутатора.

[7]Подчеркнём, что от записи защищено не внимание, а именно предыдущий коммутатор.

Таким образом, как запись на доске, так и чтение с неё могут в равной степени выполняться маркером как средством передачи информации.

Базовый сценарий полагает, что маркер записывает информацию (`argument`) без изменений. Однако, принцип универсальности требует, чтобы нарушение этого правила, т.е. изменение информации между чтением и записью, также поддерживалось. Понятие *изменения* или *преобразования* напрямую вытекает из начального постулата: “мыслю” подразумевает, что содержимое памяти может меняться.

Для поддержки изменения добавим в класс `marker` два члена:

```
1       public dynamic transform;
2       public drn direction;
```

Прог. 6: Поля описания изменения

Чтобы маркер мог изменять информацию, нужен объект, описывающий требуемое изменение. Таким объектом является поле `transform`, добавленное к коду маркера. Второе добавленное поле (`direction`) указывает на *направление* изменения. Мы исходим из того, что преобразование может иметь прямое или обратное направление (в некоторых случаях обратное преобразование отменяет прямое, и наоборот). Положительное направление соответствует прямому, отрицательное – обратному преобразованию.

Перепишем функцию `function`, как в Прог. 7.

```
1   public void function(object previous, object next) {
2
3       argument = read(previous);
4       argument = transform(argument, direction);
5
6       if (previous == next) throw new Exception();
7       write(next, argument); }
```

Прог. 7: Маркер: изменение

С. 4 выполняет `transform` в указанном направлении, и записывает результат в память маркера.

Таким образом, мы взяли за основу самоочевидные аспекты маркера в виде базового сценария и рассмотрели расширение и нарушение этого сценария. Результатом этого стала модель, представляющая математические аспекты маркера, вытекающие из начального постулата. Отметим, что данная модель в точности соответствует процессу вычисления *функции*, хотя математика и не акцентирует внимание на защите аргумента от записи.

В формулах будем обозначать маркер символом $\mathscr{M}$.

Определение маркера в графической нотации даётся в 4.10.

3.4. Ластик. Менее очевидным, но всё же необходимым определением является *ластик* $\mathscr{E}$. Его необходимость следует из того, что сама по себе запись является обращаемой операцией. Т.к. любые операции в AMM выполняются маркером, ластик является просто признаком того, что объект нужно стереть.

Ластик задаётся как пустой класс `public class eraser { }`, т.к. мы не усматриваем в нём каких-либо элементов. Чтобы выразить на C# способность ластика стирать объект, заменим последнюю строчку в коде маркера следующим кодом (Прог. 8):

```
1       if (argument is eraser) erase(next);
2       else write(next, argument);
```

Прог. 8: Ластик: стирание объекта

Например, применение ластика к доске как таковой очищает всю память.

§ 4. Абстрактные объекты

4.1. Универсальные термины. Необходимость понятия “объект” как предмета рассуждений следует из “мыслю”, и других определений не требуется. По принципу универсальности, объектом может являться всё что угодно.

Ключевым свойством объектов является их способность состоять из частей.

Так, начальный постулат является цельной фразой с законченным смыслом. В то же время, он состоит из элементов (слов). Другими словами, НП является *системой*; таким образом, необходимость термина “система” также напрямую вытекает из этой фразы.

Суть понятия системы программно выражается классом `system` (Прог. 9).

```
1   public abstract partial class system {
2       public abstract system[] parts(); }
```

Прог. 9: Система

Данный класс определяет функцию `parts()`, которая возвращает набор элементов системы. Отметим, что эта функция является иллюстративной: структура объектов, как правило, более сложная, чем это может быть однозначно выражено последовательностью. Система не обязана иметь элементы (см. 4.2), но если они есть, их должно быть более одного: единственный объект не образует *новую* систему.

В соответствии с принципом универсальности, можно также рассмотреть объект, который входит в состав своих же частей: например, записать `parts() => new[] { this, ... }`. Рекурсивные по `parts()` алгоритмы, обрабатывающие такое определение, будут зацикливаться; однако, это указывает не на некорректность самого понятия системы, а лишь на то, что несуществующие объекты отличаются в поведении от существующих (более подробно о существовании см. 5.4, 7.3).

Термины “объект” и “система” будем называть *универсальными*.

4.2. Нуль-система. Систему без элементов (пустую) будем называть *нуль-системой* (класс `empty`), см. Прог. 10.

Графическая нотация ФОРАОН изображает нуль-систему как кружочек (Рис. 2). Его размер не фиксируется и может меняться для удобства. Этот кружочек называется *примитивом*.

```
public class empty : system {
    public override system[] parts() => new system[]{}; }
```

Прог. 10: Нуль-система

Рис. 2: Примитив

В формулах будем обозначать нуль-систему символом $\underline{\alpha}$. Из определения следует, что $\underline{\alpha}$ является простейшим объектом и конечным этапом рекурсивной декомпозиции входной системы.

Мы будем выделять особенный примитив из числа входящих в состав сложного объекта, с которого начинается процесс его конструирования (он же отвечает за представление объекта в целом). Данный примитив будем называть *базисом* объекта. В программной нотации, базису объекта соответствует поле `public empty basis`, добавленное в класс `system`.

4.3. Дискриминаторы. Принципиальная функция памяти заключается в хранении объектов. А именно, память АММ должна хранить *множественные* объекты, из чего следует, что необходим способ отличать экземпляры объектов в памяти друг от друга. Такой способ будем называть *дискриминатором*.

Различные виды памяти реализуют различные способы различения объектов. Например, для памяти машины Тьюринга дискриминатором является позиция (номер ячейки) на ленте. Это позволяет в разные ячейки поместить символы и ссылаться на них и изменять их по отдельности. Аналогично, в оперативной памяти ЭВМ дискриминатором является адрес объекта.

Рассмотрим дискриминаторы на доске. Первым и самым очевидным из них является место (координаты) объекта на доске. По сути, это аналог номера ячейки, с поправкой на двухмерность доски.

Характерно, что для одного (изолированного) объекта его размещение не существенно: будь то доска, лента или память ЭВМ, не важно, где он расположен. Координаты объекта играют роль тогда, когда рассматриваются несколько объектов, взаимно расположенные в памяти тем или иным образом.

Кроме координат, доска имеет также два дополнительных дискриминатора: направления и ориентации. Отметим, что это более сложные, независимые способы отличать объекты друг от друга: действительно, одна точка не может задать ни направление, ни ориентацию.

Использование этих дискриминаторов более подробно рассмотрено ниже. Характерно, что ни лента машины Тьюринга, ни RAM не содержат дополнительных дискриминаторов, а именно ими объясняется то “богатство” математической реальности, о котором говорилось в 2.5.

Для учёта дискриминаторов, добавим новый член `place` (размещение) к классу `system` (Прог. 11):

```
1  public abstract partial class system {
2      public object place; }
```

Прог. 11: Размещение

Рассмотрим следующий код (Прог. 12). В классе `discriminator` два (в остальном одинаковых) члена различаются по именам, т.е. система имён в языке программирования представляет собой разновидность дискриминатора. Указанный абстрактный класс является основой для других определений ФОРАОН, при этом вместе с ним используются разные дискриминаторы на доске.

```
1  public abstract class discriminator : system {
2      public system o1;
3      public system o2; }
```

Прог. 12: Дискриминатор

Значения (`place` и аналогичные), связанные с дискриминаторами, будем называть *информацией*.

Завершая описание дискриминаторов, нужно отметить, что они относятся к конкретному устройству памяти. Так, мы ничего не знаем о том, какие дискриминаторы есть в нейронной сети мозга. Можно, однако, предположить, что память человека не менее богата, чем доска.

4.4. Контракт. Первым из объектов, более сложных, чем $\underline{\alpha}$, является *контракт*. Для его построения будем использовать один из дискриминаторов, а именно направления на доске. Построение контракта начнём с одиночного примитива (базис), вслед за которым запишем по одному примитиву в положительном и отрицательном направлении от базиса. Эти два примитива будем называть *полюсами* контракта.

Определение контракта в графической нотации изображено на Рис. 3. Ось, вдоль которой расположены полюсы, обозначена символом “коннектор” (короткая линия). В статье, расстояние между полюсами может меняться для удобства восприятия, но с математической точки зрения мы считаем, что это расстояние является некоторой элементарной величиной.

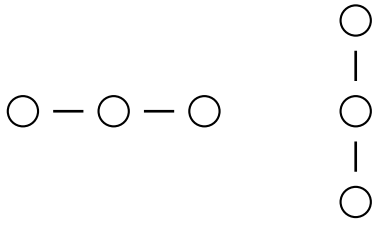


Рис. 3: Контракт (неориентированный)

Мы можем рассматривать или не рассматривать ориентацию оси, по которой размещён контракт. Если ориентация не рассматривается, то изображения по

вертикали и горизонтали эквивалентны; также допускается изображение под углом или изогнутое. Если же ориентация рассматривается, то поле `place` контракта принимает одно из значений `orn`, изображения по горизонтали и вертикали получают разный смысл, а коннектор изображается двойной линией (Рис. 4).

Рис. 4: Контракт (ориентированный)

Рассмотрим программное определение контракта (Прог. 13). Здесь `o1` и `o2` представляют полюсы, а конструктор `contract()` отражает последовательность построения объекта. Функция `parts()` возвращает его элементы: базис, положительный и отрицательный полюсы.

```
1  public class contract : discriminator {
2
3      public contract() {
4          basis = new empty();
5          (o1, o2) = (new empty(), new empty());
6          (o1.place, o2.place) = (drn.p, drn.n); }
7
8      public override system[] parts() =>
9          new[] { basis, o1, o2 }; }
```

Прог. 13: Контракт

В формулах будем обозначать контракт символом $\underline{\omega}$.

4.5. Краски. Мы будем применять дополнительные элементы графической нотации для иллюстрации расположения примитивов в контракте и соответствующих значений `place`. Эти элементы для “раскрашивания” примитивов будем называть *краски*. Поскольку применение разных цветов в статье непрактично, будем использовать черно-белые элементы, см. Рис. 5.

Рис. 5: Контракт (раскрашенный)

Ещё один вид краски рассмотрен далее в статье (см. 4.10).

4.6. Порядок. Одним из понятий, вытекающих из начального постулата, является отношение *порядка* (*следования*[8]), т.е. “предыдущий и следующий”. Необходимость порядка следует из: слова “следовательно”, упорядоченности слов в записи НП, а также предыдущего и следующего моментов времени.

[8]Не смешивать с импликацией.

Отношение порядка традиционно изображается в виде стрелочки (запись в нотации ФОРАОН см. 4.8). Запись объекта перед стрелочкой означает, что объект рассматривается, конструируется, обрабатывается или выполняется перед объектом, стоящим после стрелочки (связь порядка с временем).

Отношение порядка между двумя объектами (т.е. указание на один из них) требует одного бита информации. В программной нотации, отношение порядка будем моделировать классом `order` (Прог. 14), в котором для указания на один из объектов `o1`, `o2` используется направление (`drn`).

```
1  public abstract class order : discriminator {
2
3      public drn direction;
4      public system selected() =>
5          new[] { o1, o2 }[(int)direction]; }
```

Прог. 14: Порядок

Стрелочка имеет два смысловых наполнения. Для пары объектов, она задаёт их порядок. Кроме того, если объектов более двух, то стрелочка указывает, какие из этих объектов образуют пару.

4.7. Акцент. Традиционно, символ записывается на доске, и других вариантов не предусмотрено. В ФОРАОН, в силу принципа универсальности, примитив можно надписать как прямо на доске, так и поверх другого примитива, так что функцию памяти играет объект, отличный от доски. Данный вариант расположения примитивов имеет название *акцент* (`accent`).

Графическое определение акцента дано на Рис. 6, программное в Прог. 15.

Рис. 6: Акцент

```
1  public class accent : discriminator {
2      public override system[] parts() => new system[]{}; }
```

Прог. 15: Акцент

В данном классе нижний примитив соответствует полю `o1`, верхний полю `o2`; при этом поля `o1` и `o2` упорядочены в памяти программы, нижний и верхний примитивы также упорядочены между собой графически. Отметим, что данные поля не входят в функцию `parts()` акцента. Причина этого в том, что сам факт надписывания одного примитива над другим не состоит из элементов.

В формулах будем обозначать акцент символом $\dot{\to}$.

4.8. Директор. Другим вариантом представить порядок является стрелочка. В ФОРАОН она моделируется как указание на один из полюсов контракта. В отличие от акцента, мы рассматриваем стрелочку как изменяемый объект, т.е. она может быть инвертирована.

Построим объект, называемый *директором*. Для этого возьмём контракт и надпишем (при помощи акцента) над его базисом выбранное направление. В графической нотации это выражается как на Рис. 7i. Мы будем также использовать обобщённую форму директора для выражения следования (Рис. 7ii).

(i) Обычный (ii) Обобщённый

Рис. 7: Директор

Программная нотация задаёт определение директора как класс `director` (Прог. 16), компонентами которого являются: контракт `c` и акцент `a`. Конструктор `director()` определяет порядок конструирования директора. Функция `selected()` возвращает выбранное направление, по аналогии с `order`.

```
public class director : system {

    public contract c;
    public accent a;

    public director(drn d = drn.p) {
        c = new();
        a = new() { o1 = c.basis,
            o2 = new[] { c.o1, c.o2 }[(int)d] };
        basis = c.basis; }

    public drn selected() => (drn)a.o2.place;

    public override system[] parts() =>
        new system[] { c, a, }; }
```

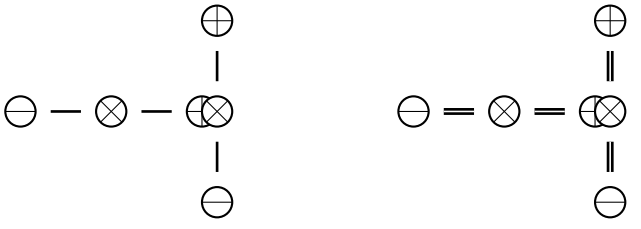
Прог. 16: Директор

В формулах будем обозначать директор символом $\mathscr{D}$ или стрелочкой.

4.9. Цепочка. Рассмотрим, как построить объект из двух (или более) контрактов. Будем объединять контракты путём надписывания (акцент) базиса следующего контракта над полюсом предыдущего. Объединяемые контракты изображаются под различными ориентациями (Рис. 8ii).

Данный объект будем называть *цепочкой* и обозначать как $\mathscr{S}$.

Программная нотация задаёт цепочку как класс `sequence` (Прог. 17). В нём объекты `o1` и `o2` это первый и второй контракты, размещение которых принимает значения ориентации, первичная и вторичная. Базисом цепочки является базис первого контракта. Элементами цепочки являются контракты



(i) Неориентированная (ii) Ориентированная

Рис. 8: Цепочка

и акцент(ы) между ними. Если следующие контракты надписаны только над одним из направлений предыдущих, такую цепочку назовём *линейной*.

```
public class sequence : discriminator {

    public accent a;
    public sequence() {

        (o1, o2) = (new contract(), new contract());
        (o1.place, o2.place) = (orn.p, orn.s);
        a = new() { o1 = ((contract)o1).o1, o2 = o2.basis };
        basis = o1.basis; }

    public override system[] parts() => new[] { o1, o2, a }; }
```

Прог. 17: Цепочка

4.10. Собственный контур. Определение собственных объектов на C# дано в 3.1. Теперь запишем графическую форму (Рис. 9), которую будем называть *собственным контуром*.

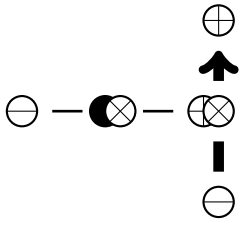


Рис. 9: Собственный контур

Построение начнём с внимания (“я”). Внимание изображается примитивом, который залит чёрной краской. Поверх внимания записывается смешанная система, которой соответствует контракт, где отрицательный полюс есть память, положительный время.

Для отражения структуры времени используем директор, надписанный над полюсом времени. Коннектор директора маркера выполнен жирной линией. Его полюсы представляют коммутаторы, базис преобразование (`transform`), выбранное направление действия маркера надписано акцентом над базисом.

4.11. Координаты. Необходимость понятия координат следует из записи начального постулата через свойства плоскости. Действительно, любая надпись на доске находится на некотором месте, а это место может находиться сверху/снизу и/или справа/слева от другого. Формализация понятия места и есть координаты.

Мы можем рассматривать любой объект как абстрагируясь от его координат, так и совместно с ними. В первом случае будем использовать обычную запись x, во втором формулу $\mathscr{B} \to x$. Данная формула уточняет, что мы рассматриваем объект на доске, где он имеет координаты. В отличие от доски, память маркера содержит всего один объект, и координаты к нему не применимы.

Координатами сложного объекта будем считать координаты его базиса.

§ 5. Темпоральная механика

5.1. Терминология. Под *транзакцией* будем понимать единичный акт действия маркера, однократное событие исполнения метода `function()`. Под *алгоритмом* будем понимать объект, соответствующий полю `transform` маркера, который описывает изменение аргумента определённым образом в процессе транзакции.

Термином *темпоральная механика* будем собирательно обозначать части низшей математики, так или иначе относящиеся к понятиям времени и маркера. Темпоральная механика интересуется феноменом и структурой времени, а также поведением объектов в роли алгоритмов.

В современной математике, наиболее близким к понятию времени является термин *функция* [1; с. 712]. Данное понятие оперирует парой множеств, между которыми существует отображение, или закон, который сопоставляет элементы множеств. Близость этих терминов обусловлена тем, что закон функции может быть записан как алгоритм, который позволит вычислять эту функцию при помощи транзакции. По этой причине транзакцию, которая читает объект x, изменяет его алгоритмом f и записывает в y, будем записывать в традиционной для функций нотации $y = f(x)$.

Состояние действий, выполняемых маркером (чтение, изменение, запись и т.д.) будем называть *моментами* времени. С действием связано минимум два момента, первый соответствует начальному состоянию действия, второй его завершённому состоянию.

5.2. Потоки и изоляция. И в физическом мире, и в АММ действия могут выполняться как независимо друг от друга, параллельно, так и вслед друг за другом. В последнем случае действия и моменты являются предыдущими/следующими в отношении друг друга.

Область действия терминов “предыдущий/следующий” и “ожидание” будем называть *потоком*. Следующий момент времени следует за предыдущим только в границах одного потока, т.е. потоки *изолированы* друг от друга. В рамках математической реальности, потоки соответствуют маркерам.

Если абстрагироваться от цепочки более простых действий внутри маркера, у последнего может быть два состояния: начальное и конечное (выполненное). Сказанное никак не меняется от того, является ли набор действий конечным, бесконечным или несчётным. В отсутствие физического времени, количество действий, необходимых для перехода из начального в конечного состояния, имеет чисто теоретический интерес; следовательно, мы можем просто рассматривать маркер в интересующем нас состоянии.

5.3. Порядок следования. Средства порождения структуры в ФОРАОН диктуют определённые правила следования. Так, базис контракта предшествует полюсам: в его отсутствие полюсы нельзя записать, т.к. их расположение было бы не определено.

Запись объекта начинается с его базиса, что вполне естественно, т.к. это единственный объект, гарантированно присутствующий в записи любой абстракции. При наличии контракта, полюсы записываются вслед за ним. После записи полюсов, появляется возможность надписать один из них над базисом, образуя директор, и/или надписать другие примитивы (контракты) над полюсами. Наконец, после записи всех примитивов, можно записывать выходящие из них стрелочки и следовать по ним.

Таким образом, структура объектов естественным образом образует граф следования примитивов. При оценке структуры объекта необходимо принимать во внимание именно граф следования, а не линейную цепочку `parts()`.

5.4. Синхронность и противоречия. Рассмотрим принципиальное отличие операций чтения и записи. Поскольку начало и конец операции отстоят во времени, то при чтении, чтобы получить данные, надо дождаться окончания операции. В отличие от чтения, запись не возвращает данные, поэтому дожидаться завершения не требуется.

Свойство операции ожидать завершения будем называть *синхронностью*[9].

Чтение и запись элементов выполняется в порядке следования (см. 5.3). В этой связи необходимо рассмотреть ситуацию, когда обнаруживается, что элемент следует за собой или более чем за одним другим элементом. Подобную ситуацию будем называть *противоречием*, или *коллапсом времени*. Она характеризуется тем, что однозначно установить прошлое объекта становится невозможным.

Запись, как асинхронная операция, движется по графу объекта от прошлого к будущему и не возвращается, поэтому она не препятствует занесению в память объектов с противоречиями (в противном случае мы не могли бы о них узнать). Однако чтение, выполняясь синхронно, движется от прошлого к будущему и обратно, поэтому невозможность вернуться по графу следования в однозначное прошлое должна приводить к аварийному завершению операции (исключение)[10].

[9] В смысле, принятом в программировании.

[10] Коллапс времени, в частности, наблюдается при равенстве коммутаторов маркера.

Надо полагать, что неясная природа парадоксов связана с тем, что противоречивое высказывание может быть формально записано, но не может быть формально прочитано. Без осмысления различий между чтением и записью, данная ситуация воспринималась как нечто невозможное.

5.5. Транзакция. Запишем транзакцию $y = f(x)$ в графической (Рис. 10) и программной (Прог. 18) нотации.

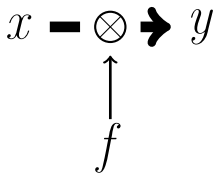


Рис. 10: Транзакция

Здесь алгоритм f записывается в поле `transform` маркера, после чего исполняется метод `function()`. Маркер не содержит ограничений в отношении объектов, которые можно использовать как алгоритм. Так, нет ограничения по дискретности: непрерывные алгоритмы равно допустимы.

На языке C#, эта транзакция сводится к вызову алгоритма f (Прог. 18):

```
1       dynamic f = ...;
2       object x = ...;
3       object y = f(x);
```

Прог. 18: Транзакция

Напомним, что мы, в общем случае, характеризуем алгоритмы направлением, положительным или отрицательным (т.е. $f^+(x)$ или $f^-(x)$). В некоторых случаях положительный и отрицательный алгоритмы взаимно обратны, в некоторых нет; а для ряда алгоритмов существует только один вариант.

5.6. Пустой алгоритм. В вырожденном случае, алгоритм не вносит никаких изменений. Такой алгоритм будем называть *пустым* (он необходим в том числе по принципу универсальности). Пустой алгоритм исполняется нуль-системой: $x \equiv \underline{\alpha}(x)$. Действительно, нуль-система не содержит ни элементов, ни информации, которые могли бы описывать изменение.

5.7. Структурные алгоритмы. Запись новых элементов объекта также называется *конструированием*. В ФОРАОН объекты имеют явно заданный порядок создания, в процессе которого, подобно зародышу, они проходят стадии более простых объектов. Например, объект сначала является единичным примитивом, затем контрактом, а затем, наконец, директором, в соответствии с графом следования.

Путём использования ластика, можно проводить обратное к конструированию *деконструирование*. Конструирование и деконструирование относятся к *структурным* алгоритмам. Другим примером таких алгоритмов являются алгоритмы над множествами (см. 8.3, 8.4).

§ 6. Движения

6.1. Обобщённое движение. Рассмотрим теперь алгоритмы, которые изменяют информационный аспект объекта. Исходя из свойств плоскости и имеющихся дискриминаторов (направления, ориентации, координаты), в наличии два алгоритма: вращение и сдвиг, вместе называемые *движениями*.

Источником необходимости движений является объединение понятия изменения с понятиями направления, ориентации и координат. Таким образом, движения вытекают из *смысла и записи* начального постулата. Оба движения характеризуются направлением и величиной; однако поначалу мы будем игнорировать величину, рассматривая некоторое элементарное значение. Сдвиг также имеет ориентацию.

В мнемонической форме движение будем записывать словом “переместить”.

Найдём объект, который служит алгоритмом движения. По структуре и информации движение точно соответствует директору, т.е. его результат вычисляется как $y = \mathscr{D}(x)$. Директоры $\mathscr{D}^+$ или $\mathscr{D}^-$ выполняют вращение (поворот), директоры $\mathscr{D}_p^+$, $\mathscr{D}_p^-$, $\mathscr{D}_s^+$ или $\mathscr{D}_s^-$ – сдвиг.

Примем это представление движения за основу и продолжим рассуждения.

6.2. Пустое движение. Принцип универсальности требует рассмотреть движение, которое не вносит изменений (пустое движение). Отметим, что пустое движение не эквивалентно пустому алгоритму. Последний является более простым понятием: пустое движение всё же является движением.

Модель пустого движения найдём “методом горшочка”. Рассмотрим директор. Он представляет собой контракт, над базисом которого надписан один из полюсов. Представим, что контракт – это горшочек, внутри которого лежит надписанный полюс[11]. Тогда пустой горшочек – это контракт, над которым полюс не надписан. Отсюда алгоритм пустого движения есть контракт.

Пустое движение, таким образом, записывается как $x \equiv \underline{\omega}(x)$; мнемоническая запись – “стоп”.

6.3. Завершённое движение. Рассмотрим одно движение в заданном направлении вслед за другим. Понятно, что комбинация элементарных движений также представляет собой движение, отличающееся от элементарного только величиной. Структуру такого движения в нотации ФОРАОН выразим цепочкой (Рис. 11i). В виде формулы такой объект записывается как:

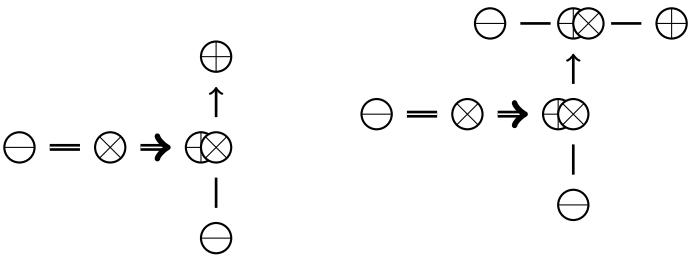
$$\mathscr{D}_p'^+ \equiv \mathscr{D}_p^+ \to \mathscr{D}^+ \tag{6.1}$$

Данная запись, при её логичности, содержит проблему, которую необходимо указать и решить. Суть проблемы – ввод паразитного смысла. Обозначим директоры как 1 и 2, а их примитивы как $\underline{\alpha}_i^0, \underline{\alpha}_i^+, \underline{\alpha}_i^-$. Тогда положительный полюс первого директора имеет формулу $\underline{\alpha}_1^+ \dot{\to} \underline{\alpha}_2^0$, в то время как второго записывается как $\underline{\alpha}_2^+$. Т.к. оба движения абсолютно идентичны, чтобы запись была верной и не содержала непреднамеренно введённого смысла, это различие необходимо устранить.

Для этого надпишем над вторым директором контракт (Рис. 11ii). Получившийся объект в мнемонической форме можно записать как “переместить,

[11] Контракт выполняет функцию памяти в отношении надписанного полюса.

переместить, стоп", в формуле – $\mathscr{D}_p^+ \to \mathscr{D}^+ \to \underline{\omega}$, что полностью соответствует смыслу указанного движения, которое будем называть *завершённым*.



(i) Незавершённое (ii) Завершённое

Рис. 11: Варианты движения

6.4. Неограниченное движение. Рассмотренное выше завершённое движение – не единственный способ устранить ввод паразитного смысла. Другим способом является применить *повторение*, как показано на Рис. 12. Здесь прерывистая линия стрелочки означает повторение во времени, т.е. цикл. Мнемоническая запись: "переместить, повторить (переместить)". В символическом виде повторение будем записывать как $\mathscr{R}$. Важно: повтор операции не означает, что объект следует сам за собой.

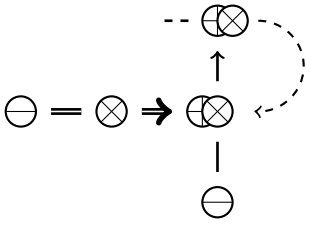


Рис. 12: Неограниченное движение

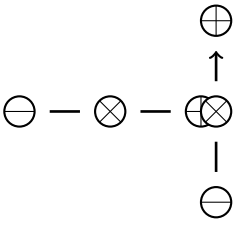
Назначение данного алгоритма – нахождение всех движений, состоящих из элементарных движений.

6.5. Редуцированное движение. Формула (6.1) задаёт новое движение в форме $a \equiv b \to b$, при этом определяемый объект (a) находится в левой части формулы. Перепишем формулу, поменяв элементарное движение и определяемый объект местами:

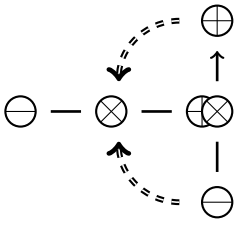
$$\mathscr{D}^+ \equiv \mathscr{D}'^+ \to \mathscr{D}'^+ \tag{6.2}$$

Движение $\mathscr{D}'$ в данной формуле будем называть *редуцированным*.

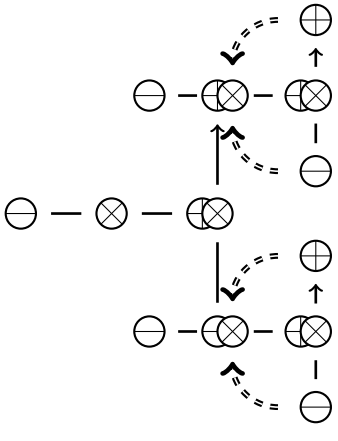
На доске выберем (произвольно) некоторую точку "стоп" (s). Поместим над ней центр движения (Рис. 13i). Ясно, что точка s делит движение пополам.



(i) Начальное



(ii) Повторение 1



(iii) Повторение 2

Рис. 13: Редуцирование

Далее повторим это действие, для чего добавим к объекту две стрелочки (Рис. 13ii). Стрелочки выполнены двойной прерывистой линией, что означает "параллельный цикл", т.е. повторы, выполняемые одновременно разными маркерами. Стрелочки предписывают далее разделить движения $\mathscr{D}'$ новыми точками. Поскольку стрелочки являются элементами объекта, при повторе действия они копируются в определение, таким образом, задавая параллельно-рекурсивный алгоритм получения новых движений $\mathscr{D}'', \mathscr{D}''', ...$, каждое из которых в два раза меньше предыдущего.

С алгоритмом редуцирования связано понятие *непрерывности*. Действительно, каждую стрелочку можно разделить на две меньших и так до бесконечности, что выражает на ФОРАОН самоподобие отрезка, возможное благодаря непрерывности доски. Непрерывность, т.е. объект со структурой на Рис. 13ii, будем обозначать символом $\mathscr{T}$.

6.6. Комбинации вращения. До настоящего момента наши рассуждения одинаково относились как к вращению, так и к сдвигу. Теперь необходимо указать принципиальное различие между ними.

При повторении сдвига в одном направлении результат никогда не становится равным пустому сдвигу. Это не верно для вращения. Если ориентация

вращаемого объекта (например, директора) не рассматривается, то повтор вращения эквивалентен пустому вращению[12]: $\mathscr{D}^+(\mathscr{D}^+) \equiv \mathscr{D}^-$, $\mathscr{D}^+(\mathscr{D}^-) \equiv \mathscr{D}^+$.

Если же ориентация рассматривается, то вращение последовательно меняет ориентацию и направления: $\mathscr{D}^+(\mathscr{D}_p^+) \equiv \mathscr{D}_s^+$, $\mathscr{D}^+(\mathscr{D}_s^+) \equiv \mathscr{D}_p^-$, $\mathscr{D}^+(\mathscr{D}_p^-) \equiv \mathscr{D}_s^-$, $\mathscr{D}^+(\mathscr{D}_s^-) \equiv \mathscr{D}_p^+$, т.е. четыре вращения в одном направлении эквивалентны пустому вращению.

6.7. Свободное движение. Из формул (6.1) и (6.2) следует, что в общем случае движение может быть любой комбинацией кратных и/или редуцированных движений, которая может быть конечной или бесконечной. Движение произвольной величины будем называть *свободным*.

Ввод свободного вращения говорит о том, что для свободного сдвига, надо заменить фиксированное перечисление `orn` в качестве ориентации контракта на непрерывное значение. Тогда значению `orn.p` будет соответствовать 0 или π, а значению `orn.s` – $\frac{1}{2}\pi$ или $\frac{3}{2}\pi$.

§ 7. Математические понятия

Мы ввели необходимые абстракции, составляющие описание математической реальности. Теперь можно связать основные математические понятия с теми или иными структурами ФОРАОН.

7.1. Объект. Само понятие объекта уже обсуждалось в 4.1. Здесь напомним, что в рамках АММ, *рассмотрение* объекта означает запись его структуры на доске маркером. Наличие описанной структуры или отыскание таковой относит предмет рассуждений к математическим объектам.

Отдельно коснёмся вопроса численных характеристик памяти и объектов. Пример человека показывает, что объём и размерность мозга не ограничивает рассмотрение разумом исчислимых и неисчислимых бесконечностей и объектов любой размерности. Это объясняется тем, что в память записывается не “сам объект” во всей своей полноте, а лишь его структура. То же самое верно и для АММ, где структура записывается на доске.

Присутствие $x(\mathscr{B})$ есть состояние объекта x после создания (применение объекта к доске), а *отсутствие* $\mathscr{E}(x)$ есть его состояние после стирания (применение ластика к объекту).

7.2. Равенство. Необходимость понятия *равенства* следует из способности мышления оценивать изменения. Проиллюстрируем равенство на следующем примере. Запишем на доске число 5 в двух местах. Единственное, что отличает эти объекты друг от друга, это их координаты, т.е. значение дискриминатора.

Под *отличием* y *от* x будем понимать алгоритм, который преобразует один объект в другой. Если этот алгоритм пуст, т.е. $y = \underline{\alpha}(x)$, то изменения y относительно x отсутствуют, что и примем за формальное определение равенства. В данном случае, дискриминаторами выступают не координаты, а сами символические обозначения x и y.

Известны аксиомы равенства, например $\forall x(x = x)$ или $\forall x \forall y(x = y \Rightarrow y = x)$ [3; с. 113]. Поскольку алгоритм $\underline{\alpha}(x)$ не вносит в x изменений, для него не

[12]Здесь директор выступает как аргументом, так и алгоритмом транзакции.

важно, какова структура x, а в случае единственного объекта x отсутствует различие даже в дискриминаторах, что доказывает первую аксиому. Если $x = y$, то различие между ними лишь в обозначениях, т.е. смысл не изменится, если переназвать x как y и наоборот: так доказываются вторая аксиома.

Знак $\equiv$, понимаемый в математике как "тождественность" мы будем всегда понимать в смысле равенства структуры, в то время как отношение $=$ может быть переопределено для конкретного вида объектов, например множеств или чисел; так, $1 + 1 = 2$, но $1 + 1 \not\equiv 2$, поскольку $1 + 1$ это операция, а 2 – число.

Запись $y \equiv x$ будем также прочитывать "y в точности является x". Если структура x входит в структуру y, но последний имеет дополнительные шаги конструирования, то будем говорить, что y является x, но не в точности. Это отношение будем называть *подобием*.

Помимо равенства, знак $=$ используется для записи присваивания: $y = f(x)$.

7.3. Существование. Описывая предел осознания (см. 3.1), мы характеризовали "я существую" как указание на собственный характер памяти и времени. Рассмотрим теперь существование для случая абстрактных объектов.

Традиционно поясняется, что объект существует, если его определение не противоречиво. Ранее, в отсутствие способа построить простейшие объекты с нуля, нельзя было проверить их непротиворечивость, которую приходилось принимать на веру.

Описав противоречие, как ситуацию следования объекта за собой или более чем одним другим объектом (коллапс времени, см. 5.4), получаем строгое определение существования. Предъявив определение объекта и показав однозначность следования, мы доказываем его существование.

7.4. Логические значения. В понятие логических значений входит: истина (T), ложь (F) и функция отрицания. Значения T и F мы полагаем атомарными, они не состоят из частей. То есть, по составу элементов и информации, логические значения эквивалентны полюсам контракта: $T \equiv \underline{\alpha}^+$, $F \equiv \underline{\alpha}^-$, отрицание эквивалентно вращению: $\neg(x) = \mathscr{D}(x)$.

Различия в смысле, делающие истину истиной, а ложь ложью, появляются при вводе функций двух логических аргументов. По числу возможных наборов результатов, имеется всего 16 различных функций, разделённые на 8 пар (алгоритм и его отрицание).

7.5. Числа. По составу элементов и информации, число эквивалентно алгоритму сдвига. Действительно, число (кроме нуля) имеет знак, величину (модуль) и угол (аргумент) к оси x на комплексной плоскости. Эти элементы соответствуют направлению, величине и ориентации сдвига.

В этой связи, можно отметить следующую терминологию и структуру:

- нуль $0 \equiv \underline{\omega}$
- единица $1 \equiv \mathscr{D}_p^+ \dot{\to} \underline{\omega}$
- мнимая единица $i \equiv \mathscr{D}_s^+ \dot{\to} \underline{\omega}$
- вещественное число $\mathbb{R}$ есть алгоритм сдвига с первичной ориентацией
- комплексное число $re^{i\varphi}$ есть алгоритм сдвига на r с ориентацией φ
- *дискретное* число $\mathscr{N}$ есть алгоритм кратного сдвига, см. (6.1)
- натуральное число $\mathbb{N}$ есть $\mathscr{N}_p^+$
- целое число $\mathbb{Z}$ есть дискретное число первичной ориентации $\mathscr{N}_p$ или нуль
- натуральная бесконечность $\infty \equiv \mathscr{R}(\mathscr{D}_p^+)$
- несчётная бесконечность есть непрерывность $\mathscr{T} \equiv \mathscr{R}^2(\underline{\omega} \dot{\to} \mathscr{D}^+)$

Знак числа будем записывать в виде верхнего индекса: $1 \equiv 1^+, -2 \equiv 2^-$.

Связь определения числа с множествами рассмотрена далее.

Рассмотрим базовые операции над числами, для чего исследуем поведение числа в качестве алгоритма. Вспомним, что сдвиг вытекает из объединения понятий изменения и координаты. Со сдвигом (числом) можно связать координату: если рассматривать сдвиг точки $(0, 0)$, то величина числа совпадает с координатой конечной точки сдвига. Следовательно, число, действуя как алгоритм над числом, меняет, по аналогии с координатой, величину числа-аргумента.

Рассмотрим транзакцию (Рис. 14).

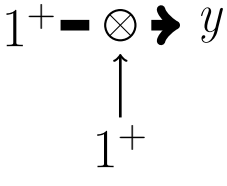


Рис. 14: $y = 1^+(1^+)$

Её алгоритм имеет единственный аргумент, т.е. $y = 1^+(1^+)$, что эквивалентно $y = 1 + 1$; в свою очередь, $1^-(1^+) = 1 - 1$. Отметим, что $x_1^+(x_2^+) = x_2^+(x_1^+)$, а $x_1^-(x_2^+) = -x_2^-(x_1^+)$. Алгоритм, симметричный в положительном направлении и асимметричный в отрицательном направлении, является повторяющимся шаблоном, который назовём *симметрической парой*.

Ранее, отсутствие структуры чисел приводило к необходимости принимать аксиомы о них на веру; например, на вопрос, является ли 1 натуральным числом, не было доказанного ответа. В нашем случае, структура чисел известна, поэтому аксиомы о числах превращаются в доказанные высказывания.

Приведём примеры.

1 *есть натуральное число.* Это высказывание следует из определения $\mathbb{N}$.

Число, следующее за натуральным, тоже является натуральным. Аксиомы Пеано рассматривают функцию следования так же, как и сами натуральные числа: как чёрный ящик. Поскольку функция следования $S(n)$ совпадает с $1^+(n)$, эта аксиома становится доказанным высказыванием.

1 *не следует ни за каким натуральным числом.* Исходя из природы функции следования, 1 следует за нулём. Т.к. нуль не входит в определение $\mathbb{N}$, единица не следует за натуральным числом.

Если натуральное число n следует как за числом i, так и за числом k, то i и k равны. Если $S(n) = 1^+(n)$, то обратная функция $S^-(n) = 1^-(n)$; значит, $i = S^-(n), k = S^-(n)$, следовательно, $i = k$.

§ 8. Множества

8.1. Природа понятия. Математическая энциклопедия [2; с. 758-759] сообщает, что "понятие множества принадлежит к числу первоначальных математических понятий и может быть пояснено только при помощи примеров". По понятным причинам, в случае АММ этого недостаточно.

Понятие множества, безусловно, связано с его элементами; но в отношении самих элементов ничего утверждать нельзя: их может не быть вовсе, или быть конечное или бесконечное количество. Отсюда, природа множества связана не с составом его элементов; но тогда с чем?

Поскольку мы начинаем конструирование объекта с одиночной нуль-системы, множество неминуемо должно конструироваться *ранее своих элементов.* Данное утверждение, по сути, единственное, что достоверно известно о любом множестве. Иными словами, множество нужно рассматривать как проявление отношения порядка/следования.

Поэтому: в точности множество есть директор $\mathscr{D}$. Объект, стоящий до стрелочки, будем называть *телом* множества, объект(ы) после стрелочки суть его элемент(ы), а саму стрелочку назовём *маршрутом* множества. В простейшем случае (изолированный директор) и тело, и единственный элемент множества есть нуль-система $\underline{\alpha}$.

По методу горшочка, пустое множество суть контракт: $\varnothing \equiv \underline{\omega}$.

Отсюда натуральные числа выражаются через множества: $0 \equiv \varnothing, 1^+ \equiv \{\varnothing\}, 2^+ \equiv \{\{\varnothing\}\}, 3^+ \equiv \{\{\{\varnothing\}\}\}$. Отметим, что данное выражение не является способом построить множество с n элементами, как например два элемента $\{\varnothing, \{\varnothing\}\}$, три элемента $\{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}$, и не противоречит этому способу.

8.2. Элементный состав. О множестве можно сказать, что оно выполняет функцию памяти в отношении своих элементов. Об этом говорит то, что множество "содержит" элементы, которые, в свою очередь, могут быть добавлены в него или удалены. Этот факт иллюстрирует процесс возникновения новых видов памяти.

Мы рассмотрели множество с одним элементом. Для построения множества с n элементами нужен более сложный маршрут, и примером такого маршрута является натуральное число n, где базис i-го директора цепочки связан стрелочкой с i-м элементом (Рис. 15).

Иным способом записывается множество точек отрезка. Здесь телом множества является начальная точка s: действительно, знание, где нужно записать сдвиг, предшествует собственно записи сдвига. Отношение следования (маршрут), вместо директора, записано акцентом. За телом множества следует записанный сдвиг (линия), содержащий в себе все точки.

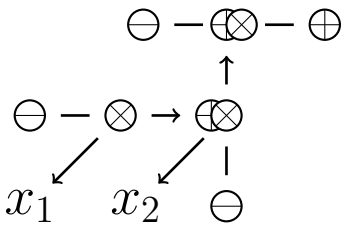


Рис. 15: Множество с 2 элементами

В связи с записью множества, надо отметить тот факт, что и доска, и лента машины Тьюринга упорядочены. По этой причине, любая *запись* множества также упорядочена. Однако, если рассматривается формирующий множество алгоритм в начальном состоянии, то результат ещё не записан, и маршрута ещё нет; есть лишь намерение в отношении состава элементов. По этой причине, множества по умолчанию рассматриваются как неупорядоченные.

Запись $\{x_1, x_2\}$ (тело не указано) эквивалентна $\underline{\alpha} \to \{x_1, x_2\}$.

Рассмотрим *универсум*, т.е. множество всех объектов. В обычном понимании, такое множество не должно существовать, т.к. следует само за собой. Низшая математика позволяет разрешить это противоречие. Любой объект, если он рассматривается, расположен в памяти. Следовательно, в качестве коллекции всех объектов естественным образом выступает память $\mathscr{B}$ как таковая. Поскольку память является не абстрактным, а собственным объектом, её существование следует из начального постулата. Поэтому, хотя память и не содержит саму себя, она содержит все абстрактные объекты.

8.3. Алгоритмы. Подобно доске, где объект может быть записан или стёрт под воздействием маркера, элемент может быть добавлен в множество или удалён из него. Добавление элемента x в множество A будем записывать как $A \oplus x$, удаление как $A \ominus x$. Если объект присутствует (отсутствует) в множестве, то добавление (удаление) не изменяет это множество, поэтому $A \oplus x$ и $A \ominus x$ не являются взаимно обратными.

Пусть A и B – множества. Рассмотрим алгоритмы $B^+(A)$ и $B^-(A)$. Применение множества B к множеству A в положительном направлении добавляет все элементы B к A, а в обратном направлении удаляет все элементы B из A. Запишем сказанное в виде формул (8.1) и (8.2). Здесь запись $(f) \to a$ будем читать как “выполнить действие f и положить a как результат”. Запись $\forall x \in A(f(x))$ будем читать как “для каждого элемента x из A выполнить действие $f(x)$”.

$$B^+(A) \equiv (C = A \quad \forall x \in B(C \oplus x)) \to C \tag{8.1}$$

$$B^-(A) \equiv (C = A \quad \forall x \in B(C \ominus x)) \to C \tag{8.2}$$

Более привычная бинарная форма записи этих алгоритмов выглядит как объединение $A \cup B$ и разность $A \setminus B$. Видно, что эти алгоритмы образуют симметрическую пару. Отметим, что данные операции не являются взаимно обратными.

Дополнение множества выразим как $\overline{A} \equiv A^{-}(\mathscr{B})$.

Перейдём к алгоритмам двух аргументов. Простейшими из них являются пересечение $A \cap B$ (8.3) и симметрическая разность $A \triangle B$ (8.4). Данные алгоритмы устроены аналогично: на основе $\oplus$ и $\ominus$, соответственно. Запись $a(x) \Rightarrow f(x)$ в данном контексте будем читать как "если функция $a(x)$ истинна, то выполнить действие $f(x)$".

$$\begin{aligned} A \cap B \equiv (&C = B^{+}(A) \\ &D = \varnothing \\ &\forall x \in C((x \in A \wedge x \in B) \Rightarrow D \oplus x)) \to D \end{aligned} \tag{8.3}$$

$$\begin{aligned} A \triangle B \equiv (&C = B^{+}(A) \\ &\forall x \in C((x \in A \wedge x \in B) \Rightarrow C \ominus x)) \to C \end{aligned} \tag{8.4}$$

Нам понадобится условие равенства множеств $(A = B)$, поэтому зададим его строго формулой (8.5).

$$\begin{aligned} (A = B) \equiv (&C = A, D = B \\ &\forall x \in A(D \ominus x) \\ &\forall x \in B(C \ominus x)) \to (C \equiv \varnothing \wedge D \equiv \varnothing) \end{aligned} \tag{8.5}$$

8.4. Аксиомы теории множеств. Описав структуру и поведение множеств, обнаруживаем, что аксиомы о них могут быть доказаны. Проиллюстрируем это на примерах (формулировка аксиом по [3; с. 126]).

Аксиома объёмности. *Любые два множества x и y, содержащие одни и те же элементы, равны:* $\forall x \forall y(\forall z(z \in x \equiv z \in y) \Rightarrow x = y)$. В соответствии с формулой (8.5), в этом случае между множествами отсутствуют относительные изменения, что эквивалентно равенству.

Аксиома пустого множества. *Существует множество $\varnothing$, не содержащее элементов:* $\exists \varnothing \forall y(y \notin \varnothing)$. Истинно по определению: $\varnothing \equiv \underline{\omega}$.

Аксиома пары. *Для любых множеств x и y существует множество z, содержащее в точности элементы x и y:* $\forall x \forall y \exists z \forall w(w \in z \equiv (w = x \vee w = y))$. Запишем транзакцию:

$$\begin{aligned} z = (&A = \varnothing \\ &A \oplus x \\ &A \oplus y) \to A \end{aligned}$$

Данная формула непротиворечиво определяет z, следовательно z существует.

Аксиома объединения (суммы). *Для любого семейства множеств x существует множество u, элементами которого являются в точности элементы множеств, принадлежащих x:* $\forall x \exists u \forall y(y \in u \equiv \exists z(z \in x \wedge y \in z))$.

Запишем транзакцию:

$$u = (A = \varnothing \\ \forall B \in x(\forall i \in B(A \oplus i))) \to A$$

Данная формула непротиворечиво определяет u, следовательно, u существует.

Аксиома множества всех подмножеств. *Для любого множества x существует множество y, элементами которого являются все подмножества множества x и только они:* $\forall x \exists y \forall z(z \in y \equiv z \subseteq x)$. Найдём способ вычисления y. Зададим некоторое множество A и парный к нему рекурсивный алгоритм Δ_A:

$$\Delta_A(a) \equiv (\forall i \in a(B = a \\ B \ominus i \\ B \not\equiv \varnothing \Rightarrow A \oplus B \\ \Delta_A(B)))$$

Тогда транзакция вычисления y запишется как:

$$y = (A = \varnothing \\ \Delta_A(x)) \to A$$

Эта формула непротиворечиво определяет y, следовательно y существует.

Схема аксиом выделения. *Для любого множества x существует множество y, содержащее в точности элементы множества x, которые выполняют условие A:* $\forall \forall x \exists y \forall z(z \in y \equiv (z \in x \wedge A(z)))$. Запишем транзакцию:

$$y = (B = \varnothing \\ \forall i \in x(A(i) \Rightarrow B \oplus i)) \to B$$

Данная формула непротиворечиво определяет y, следовательно y существует.

§ 9. Заключение

Новизна и идеологическая основа данной работы – введение формальной модели мышления (АММ) в пространство математики. Данный приём позволил предъявить структуру основных математических объектов, ранее задававшихся только аксиоматически, и описать их поведение. Обнаружилось, что эти объекты входят друг в друга как элементы, в порядке увеличения сложности:

- нуль-система $\underline{\alpha}$
- контракт $\underline{\omega}$ (логические значения, нуль, пустое множество)
- директор $\mathscr{D}$ (множества)
- цепочка $\mathscr{D} \to \underline{\omega}$ (числа)

Существенно, что данная иерархия не придумана, а обнаружена посредством анализа самодоказывающего высказывания – начального постулата. Если ранее этот тезис был предметом преимущественно философской дискуссии, то анализ его записи, наряду со смыслом, делает его исчерпывающим источником математического знания, потенциал которого демонстрирует статья.

Помимо фундаментальных математических результатов, статья позволяет строго представить смысл математического знания в системах искусственного интеллекта. Если человек приходит к пониманию числа как “объекта, возникающего в процессе счёта” постепенно, в процессе опыта, то система на базе АММ может иметь идеи числа, множества и др. понятными уже на базовом уровне своей конструкции. Также АММ содержит абстракции, требуемые для осознания системой ИИ самой себя.

Список литературы

[1] *Математическая энциклопедия*, **5**, Советская энциклопедия, Москва, 1982.

[2] *Математическая энциклопедия*, **3**, Советская энциклопедия, Москва, 1982.

[3] А. С. Герасимов, *Курс математической логики и теории вычислимости: учебное пособие. 3-е изд., испр. и доп.*, ЛЕМА, С-Петербург, 2011.

А. В. Самойлов (Andrei Samoylov)
Москва
E-mail: as@mixed.systems

16 сентября 2023 г.