Математическое введение в декларативное программирование

Оглавление

Основной вклад в развитие программирования как дисциплины несомненно внесли математики. В недрах математической логики были найдены математически точные понятия алгоритма и вычислимой функции, развита семантика формальных языков и теорий, построены системы логического вывода – и все это, заметим, было сделано в 30–40-х годах, т. е. в «докомпьютерную эру». Программирование также имеет дело с формальными языками – языками программирования. Чтобы сделать эти языки удобными и естественными для человека полезно использовать опыт математической логики, в рамках которой создан формальный язык математики. Также материальная основа программирования – современный компьютер – есть воплощение модели, предложенной математиком фон Нейманом.

Несомненно, из математиков получаются, как правило, неплохие программисты, поскольку математика подобно гимнастике вырабатывает способности, в данном случае к интеллектуальной деятельности.

Тем не менее, достаточно мало математиков, которые занимаются практическим программированием. В настоящее время существуют несколько программных систем, например, Maple и Mathematica, которые можно рассматривать как великолепные компьютерные инструменты для научных исследований в области математики. Почему же математики не стремятся овладеть этими инструментами? Может быть, все дело в том, как математики изучают программирование?

1 Что такое декларативное программирование?

> Я с легкостью сделаю на C++ за месяц то, что вы с трудом напишите на Лиспе за неделю.
>
> Программистский фольклор

1.1 Машина фон Неймана и процедурное программирование

Материальная основа программирования – современный компьютер – есть воплощение модели, предложенной математиком фон Нейманом.

Что такое компьютер фон Неймана? Когда фон Нейман и другие задумывали его более 50 лет назад, это была изящная, практичная и объединяющая идея, которая упрощала ряд существовавших тогда инженерных и программистских задач. Хотя условия, породившие архитектуру этого компьютера, с тех пор радикально изменились, тем не менее, мы по-прежнему идентифицируем понятие «компьютера» с этой концепцией пятидесятилетней давности.

В своей простейшей форме компьютер фон Неймана состоит из трех частей: центрального процессорного устройства, памяти и соединительной шины, которая может за один шаг передавать только одно слово между процессором и памятью (и посылать некий адрес в память). Можно назвать (Дж. Бэкус) эту шину «*бутылочным горлышком*» (*узким местом*) фон Неймана. В памяти размещается программа и данные, с которыми эта программа работает.

Задача программы состоит в том, чтобы неким существенным образом изменить содержимое памяти; если считать, что эта задача должна быть выполнена исключительно перекачиванием через узость фон Неймана, то становится ясной причина такого названия.

Ирония ситуации состоит в том, что большую часть потока через эту узость составляют не полезные данные, а всего лишь имена данных, а также операции и данные, служащие лишь для вычисления таких имен. Прежде чем слово можно будет послать через шину, его адрес должен находиться в процессоре; поэтому он должен либо быть послан через шину из памяти, либо генерироваться посредством некоторой операции процессора. Если адрес посылается из памяти, то адрес этого адреса должен либо быть послан из памяти, либо генерироваться в процессоре, и т. д. С другой стороны, если адрес генерируется в процессоре, он должен генерировать либо по фиксированному правилу (на-

пример, «добавить 1 к счетчику исполняемых команд»), либо по команде, которая была послана через шину, в последнем случае ее адрес нужно было прежде послать... и т. д.

Итак, программирование на традиционных языках, в основном сводиться к планированию и спецификации огромного потока слов через узость фон Неймана, причем большая часть этого потока состоит не из самих значащих данных, а из сведений о том, где их искать.

Обычные языки программирования в основном являются высокоуровневыми, сложными версиями компьютера фон Неймана. Такие языки называются процедурными; к ним относятся такие популярные языки, как Паскаль и С (С++).

Несмотря на все свою интеллектуальную привлекательность программирование является трудным занятием. После сорока лет многие перестают программировать под предлогом, что устали. Основной причиной такого положения является субъективное чувство «нарушения энергетического баланса» – много тратят, но мало получают. Программисты тратят много лет и усилий, чтобы научиться программировать в императивном (процедурном) стиле. Но императивный стиль программирования – стиль, в котором выполняются программы на компьютере – совершенно не свойственен человеческой природе. Традиционное процедурное программирование приучают программиста мыслить понятиями «машины фон Неймана», а не математическими понятиями. Традиционный программист – по своим знаниям, в первую очередь, должен быть инженером.

Процедурное программирование наиболее пригодно для решения задач, в которых последовательное исполнение каких-либо команд является естественным. Примером здесь может служить управление современными аппаратными средствами. Поскольку практически все современные компьютеры императивны, эта методология позволяет порождать достаточно эффективный исполняемый код. С ростом сложности задачи процедурные программы становятся все менее и менее читаемыми. Программирование и отладка действительно больших программ (например, компиляторов), написанных исключительно на основе методологии императивного программирования, может затянуться на долгие годы.

1.2 От процедурного к декларативному программированию

Процедурные языки имеют одну общую черту: базовый оператор в них – это оператор присваивания, который заставляет компьютер переместить данные из одного места в другое. Принципиально иной подход к программированию заключается в том, чтобы описывать вычисление, используя уравнения, функции, логические выводы и т. п. Особое внимание в декларативном программировании уделяется тому, ***что*** нужно сделать, а не тому, ***как*** это нужно сделать.

Декларативное программирование обычно применяется для решения тех задач, которых трудно сформулировать в терминах последовательных операций. В эту категорию попадают практически все задачи, связанные с искусственным интеллектом. Это такие задачи, как обработка естественного языка, экспертные консультирующие системы, проблемы зрительного восприятия, и многие другие.

Мы рассмотрим особенности двух ветвей декларативного программирования: **функциональное**, основанное на математическом понятии функции, которая не изменяет свое окружение, в отличии от функций в процедурных языках, допускающих побочные эффекты, и **логическое**, в котором программы выражены в виде формул математической логики и компьютер для решения задачи пытается вывести логические следствия из них.

1.3 Почему следует изучать декларативное программирование?

О том, чем определяется уровень программирования

Интуитивно любой программист отличит язык программирования высокого уровня от языка программирования низкого уровня. В чем состоит различие? Чем определяется уровень? Программирование представляет собой отображение в программах объектов,

понятий и явлений природной области задачи. Чем более адекватно можно выполнить это отображение, тем выше уровень языка программирования.

Декларативное программирование требует высокого уровня абстрагирования. Но повышение уровня абстрагирования необходимое требование для программиста. Эдсгер Дейкстра (Edsger Dijkstra) [3] подчеркивал, что программист должен обладать умением абстрагировать: «Язык программирования – это лишь средство описания абстрактных конструкций. Программист должен иметь способность полностью абстрагироваться от несущественных деталей, думая на нескольких уровнях абстракции одновременно».

Декларативные языки при классификации по степени абстракции от аппаратуры относятся к языкам *сверхвысокого уровня*. Команды исполняются на полностью абстрактной машине, полностью скрыт доступ к памяти, и возможно скрыть поток управления.

Концептуальная целостность языка – характеризуется свойствами совокупности понятий, служащих для описания этого языка и включает три взаимосвязанных аспекта.

- *Экономия понятий* – язык должен достигать своей максимальной мощности минимальным количеством понятий.
- *Ортогональность понятий* – между понятиями не должно быть взаимного влияния. Если понятие используется в различных контекстах, то правило его использования должно быть одним и тем же.
- *Единообразие понятий* – требование согласованного единого подхода к описанию и использованию всех понятий.

Декларативные языки являются, как правило, концептуально целостными. Язык Haskell в этом отношении является стандартом.

О неортогональности понятий даже самых лучших языков

К сожалению, даже такие языки, как Паскаль, допускают неортогональные конструкции. Например, пользователь может определить процедуры только с фиксированным числом параметров, однако некоторые стандартные процедуры (например, write) могут быть вызваны с переменным числом параметров.

Очень важны для применения языка следующие характеристики:

- *Надежность* – язык должен обеспечивать минимум ошибок при написании программ. Более того, язык должен быть таким, чтобы неправильные программы было трудно писать.
- *Удобочитаемость* – легкость восприятия программ человеком. Эта характеристика особенно важна при коллективной работе, когда несколько человек работают с одними и теми же текстами программ.
- *Полнота* – характеризует способность описать класс задач в некоторой предметной области.
- *Гибкость* – характеризует легкость выражения необходимых действий.

Ниже будет показано, что современные чистые функциональные языки обладают всеми вышеперечисленными характеристиками.

Международные организации, занимающие стандартизацией, ACM (The Association for Computing Machinery) и IEEE (Institute of Electrical and Electronic Engineers) рекомендуют в учебных программах по программированию значительное место отводить изучению различных парадигм программирования, в том числе и изучению декларативного программирования.

Точка зрения профессионалов. Тимоти Летбридж (Timothy C. Lethbridge) в 1998 году провел опрос более 200 опытных программистов [14]. Было выбрано 75 предметных областей, составляющих информатику и по которым есть учебные курсы практически в каждом высшем учебном заведении. Относительно каждой области были заданы вопросы.

- Насколько полезным оказалось знание *подробностей* именно этого предметного материала в Вашей карьере разработчика программного обеспечения или менеджера программных проектов?
- Какое *влияние* оказало изучение этого предметного материала на Ваше мышление, независимо от непосредственного применения подробностей на практике?

Пятерку наиболее полезных областей программирования составили (в порядке убывания их значимости для опрашиваемых):

- конкретные языки программирования;
- структуры данных;
- проектирование программ и паттерны;
- архитектура программного обеспечения;
- методы анализа и проектирования.

Как видно из результатов опроса профессиональное мастерство программиста во многом формируется знанием конкретных языков программирования.

2 Формальные аксиоматические теории

Формальная теория представляет собой множество чисто абстрактных объектов (не связанных с внешним миром), в которой представлены правила оперирования множеством символов в чисто синтаксической трактовке без учета смыслового содержания (или семантики).

Исторически понятие формальной теории было разработано в период интенсивных исследований в области оснований математики для формализации собственно логики и теории доказательства. Сейчас этот аппарат широко используется при создании специальных исчислений для решения конкретных прикладных задач. Одним из таких приложений является логическое программирование.

Формальную теорию иногда называют *аксиоматикой* или *формальной аксиоматической теорией*. Родоначальником аксиоматических теорией можно считать «Начала» Евклида.

2.1 Определение формальной теории

Формальная теория *T* считается определенной, если:

- задано некоторое счетное множество ***A*** символов – символов теории ***T***; конечные последовательности символов теории ***T*** называются *выражениями* теории ***T*** (множество выражений обозначают через ***A****);
- имеется подмножество $\boldsymbol{F} \subset \boldsymbol{A}^*$ выражений теории ***T***, называемых *формулами* теории ***T***;
- выделено некоторое множество $\boldsymbol{B} \subset \boldsymbol{F}$ формул, называемых *аксиомами* теории ***T***;
- имеется конечное множество $\{R_1, R_2, \ldots, R_m\}$ отношений между формулами, называемых *правилами вывода*. Правила вывода позволяют получать из некоторого конечного множества формул другое множество формул.

Множество символов ***A*** – алфавит теории – может быть конечным или бесконечным. Обычно для образования символов используют конечное множество букв, к которым, если нужно, приписывают в качестве индексов натуральные числа.

Множество формул ***F*** обычно задается индуктивным определением, например, с помощью формальной грамматики. Как правило, это множество бесконечно. Множества ***A*** и ***F*** в совокупности определяют *язык формальной теории*.

Множество аксиом ***B*** может быть конечным или бесконечным. Если множество аксиом бесконечно, то, как правило, оно задается с помощью конечного множества *схем аксиом* и правил порождения конкретных аксиом из схемы аксиом. Обычно аксиомы делятся на два вида: *логические* аксиомы (общие для целого класса формальных теорий) и *нелогические* (или *собственные*) аксиомы (определяющие специфику и содержание конкретной теории).

Обычно для формальной теории имеется алгоритм, позволяющий по данному выражению определить, является ли оно формулой. Точно также, чаще всего существует алгоритм, выясняющий, является ли данная формула теории ***T*** аксиомой; в таком случае ***T*** называется *эффективно аксиоматизированной* теорией.

2.2 Выводимость

Пусть $A_1, A_2, \ldots, A_n, A$ – формулы теории ***T***. Если существует такое правило вывода R, что $< A_1, A_2, \ldots, A_n, A> \in R$, то говорят, что формула A *непосредственно выводима* из формул $A_1, A_2, \ldots, A_n$ по правилу вывода R. Обычно этот факт записывают следующим образом:

$\frac{A_1, A_2, ..., A_n}{A} R$,

где формулы A_1, A_2,..., A_n называются *посылками*, а формула A – *заключением*.

Выводом формулы A из множества формул Γ в теории ***T*** называется такая последовательность F_1, F_2,..., F_k , что $A = F_k$, а любая формула F_i ($i < k$) является либо аксиомой, либо $F_i \in \Gamma$, либо непосредственно выводима из ранее полученных формул $F_{j_1}, ..., F_{j_n}$ ($j_1, ..., j_n < i$). Если в теории ***T*** существует вывод формулы A из множества формул Γ, то это записывается следующим образом:

$\Gamma \vdash_{\boldsymbol{T}} A$,

где формулы из Γ называются *гипотезами* вывода. Если теория ***T*** подразумевается, то её обозначение обычно опускают.

Если множество Γ конечно: $\Gamma = \{B_1, B_2, ..., B_n\}$, то вместо $\{B_1, B_2, ..., B_n\} \vdash A$ пишут $B_1, B_2, ..., B_n \vdash A$. Если Γ есть пустое множество ∅, то A называют теоремой (или *доказуемой* формулой) и в этом случае используют сокращенную запись $\vdash A$ («A есть теорема»).

Отметим, что в соотношении {теоремы}⊂{формулы}⊂{выражения} включение множеств является строгим.

Приведем несколько простых свойств понятия выводимости из посылок.

1. Если $\Gamma \subseteq \Sigma$ и $\Gamma \vdash A$, то $\Sigma \vdash A$.

Это свойство выражает тот факт, что если A выводимо из множества гипотез Γ, то оно остается выводимым, если мы добавим к Γ новые гипотезы.

2. $\Gamma \vdash A$ тогда и только тогда, когда в Γ существует конечное подмножество Σ для которого $\Sigma \vdash A$.

Часть «тогда» утверждения 2 вытекает из утверждения 1. Часть «только тогда» этого утверждения очевидно, поскольку всякий вывод A из Γ использует лишь конечное число гипотез из Γ.

3. Если $\Sigma \vdash A$ и $\Gamma \vdash B$ для любого B из множества Σ, то $\Gamma \vdash A$.

Смысл этого утверждения прост: если A выводимо из Σ и любая формула из Σ выводима из Γ, то A выводима из Γ.

Понятие формальности можно определить в терминах теории алгоритмов: теорию ***T*** можно считать формальной, если построен алгоритм (механически применяемая процедура вычисления) для проверки правильности рассуждений с точки зрения принципов теории ***T***. Это значит, что если некто предлагает математический текст, являющийся, по его мнению, доказательством некоторой теоремы в теории ***T***, то, механически применяя алгоритм, мы можем проверить, действительно ли предложенный текст соответствует стандартам правильности, принятым в ***T***. Таким образом, стандарт правильности рассуждений для теории ***T*** определен настолько точно, что проверку его соблюдения можно передать вычислительной машине (следует помнить, что речь идет о **проверке правильности** готовых доказательств, а не об их поиске!). Если проверку правильности доказательств в какой-либо теории нельзя передать вычислительной машине, и она доступна в полной мере только человеку, значит, еще не все принципы теории аксиоматизированы (то, что мы не умеем передать машине, остается в нашей интуиции и «оттуда» регулирует наши рассуждения).

В качестве несерьезного примера формальной теории можно рассматривать игру в шахматы – назовем это теорией ***Ch***. Формулами в ***Ch*** будем считать **позиции** (всевозможные расположения фигур на доске вместе с указанием «ход белых» или «ход черных»). Тогда аксиомой теории ***Ch*** естественно считать **начальную позицию**, а правилами вывода – **правила игры**, которые определяют, какие ходы допустимы в каждой позиции. Правила позволяют получать из одних формул другие. В частности, отправляясь от нашей единственной аксиомы, мы можем получать теоремы ***Ch***. Общая характеристика теорем ***Ch*** состоит, очевидно, в том, что это – всевозможные позиции, которые могут получиться, если передвигать фигуры, соблюдая правила.

В чем выражается формальность теории ***Ch***? Если некто предлагает нам «математический текст» и утверждает, что это – доказательство теоремы *A* в теории ***Ch***, то ясно, что речь идет о непроверенной записи шахматной партии, законченной (или отложенной) в позиции *A*. Проверка не является, однако, проблемой: правила игры сформулированы настолько точно, что можно составить программу для вычислительной машины, которая будет осуществлять такие проверки. (Еще раз напомним, что речь идет о проверке правильности записи шахматной партии, а не о проверке того, можно ли заданную позицию получить, играя по правилам, – эта задача намного сложнее!)

Несколько серьезнее другой пример формальной теории. Пусть {*a*, *b*} есть алфавит теории ***L***. Формулами в теории ***L*** являются всевозможные цепочки, составленные из букв *a*, *b*, например *a*, *aa*, *aba*, *abaab*. Единственной аксиомой ***L*** является цепочка *a,* наконец, в ***L*** имеется два правила вывода:

$$\frac{X}{Xb} \quad и \quad \frac{X}{aXa}.$$

Такая запись означает, что в теории ***L*** из цепочки *X* непосредственно выводятся *Xb* и *aXa*. Примером теоремы ***L*** является цепочка *aababb*; вывод для нее есть

a, *ab*, *aaba*, *aabab*, *aababb.*

Формальные теории являются не просто игрой ума, а всегда представляют собой модель какой-то реальности (либо конкретной, либо математической). Вначале математик изучает реальность, конструируя некоторое абстрактное представление о ней, т. е. некоторую формальную теорию. Затем он доказывает теоремы этой формальной теории. Вся польза и удобство формальных теорий как раз и заключается в их абстрагировании от конкретной реальности. Благодаря этому одна и та же формальная теория может служить моделью многочисленных конкретных ситуаций. Наконец, он возвращается к исходной точке всего построения и дает интерпретацию теорем, полученных при формализации.

При изучении формальных теорий нужно различать теоремы формальной теории и теоремы *о* формальной теории, или *метатеоремы*. Это различие не всегда явно формализуется, но всегда является существенным.

Множество теорем формальной теории является точно определенным объектом (обычно бесконечным) и поэтому можно доказывать утверждения, относящиеся ко всем теоремам одновременно. Например, в теории ***Ch*** множество всех теорем оказывается, правда, конечным (хотя конечность эта с практической точки зрения ближе к бесконечности). Легко доказать следующее утверждение, относящееся ко *всем* теоремам ***Ch***: ни в одной теореме белые не имеют 10 ферзей. В самом деле, достаточно заметить, что в аксиоме ***Ch*** белые имеют 1 ферзь и 8 пешек и что по правилам игры белым ферзем может стать только белая пешка. Остальное решает арифметика: 1+8<10. Таким образом, мы подметили в системе аксиом и правил вывода теории ***Ch*** особенности, которые делают справедливым наше общее утверждение о теоремах ***Ch***.

Аналогичные возможности имеем в случае теории ***L***. Можно доказать (с помощью математической индукции), например, следующее утверждение, относящееся ко всем теоремам ***L***: если *X* – теорема, то *aaX* – тоже теорема.

Рассмотрим как соотносятся неформальные доказательства и логический вывод. Логический вывод напоминает процесс мышления, но при этом мы не должны равнять его правила с правилами человеческой мысли. Доказательство – это нечто неформальное; иными словами – это продукт нормального мышления, записанный на человеческом языке и предназначенный для человеческого потребления. В доказательствах могут использоваться всевозможные сложные мыслительные приемы и, хотя интуитивно, они могут казаться верными, можно усомниться в том, возможно ли доказать их логически. Именно

поэтому мы нуждаемся в формализации. Вывод – это искусственное соответствие доказательства: его назначение – достичь той же цели, на этот раз с помощью логической структуры, методы которой не только ясно выражены, но и очень просты.

Обычно формальный вывод бывает крайне длинен по сравнению с соответствующей «естественной» мыслью. Это, конечно, плохо – но это та цена, которую приходится платить за упрощение каждого шага. Часто бывает, что вывод и доказательство «просты» в дополнении друг к другу. Доказательство просто в том смысле, что каждый шаг «кажется правильным», даже если мы и не знаем точно, почему; логический вывод прост, потому что каждый из мириада её шагов так прост, что к нему невозможно придраться и, поскольку весь вывод состоит из таких шагов, мы предполагаем, что он безошибочен. Каждый тип простоты, однако, привносит свой тип сложности. В случае доказательств, это сложность системы, на которую они опираются – а именно, человеческого языка; в случае логического выводов, это их астрономическая длина, делающая их почти невозможными для понимания.

Таким образом, мы считаем логический вывод частью общего метода для составления искусственных структур, подобных доказательствам. Однако он лишен гибкости или всеобщности, поскольку предназначен только для работы с математическими понятиями, которые, в свою очередь, жестко определены.

В качестве средства общения, открытия, фиксации материала никакой формальный язык не способен конкурировать со смесью национального математического арго и формул, привычной для каждого работающего математика.

Однако в силу своей жесткой нормализованности формальные тексты могут сами служить объектом математического исследования. Метатеоремы вызывают значительный интерес (и сильные эмоции), будучи интерпретированы расширительно (как теоремы о математике). Но именно возможность таких и еще более расширительных толкований определяет общефилософское и общегуманитарное значение математической логики.

2.3 Язык первого порядка

2.3.1 Предметы и универсум

Любое математическое утверждение в конечном счете говорит о предметах (объектах). Каждая математическая теория имеет свою *предметную область*, или *универсум*, – совокупность всех предметов, которые она изучает.

Например, универсумом теории чисел является множество натуральных чисел, а ее объектами сами натуральные числа.

Математическая теория может иметь несколько универсумов. В этом случае теория является многосортной, объекты делятся на *типы*, или *сорта*, и для каждого сорта задается свой универсум. Примерами могут служить современные языки программирования, или математический анализ – два универсума: универсум чисел и универсум функций.

Простейшие из выражений, обозначающих предметы, – *константы*, т. е. имена конкретных объектов. Например, константами служат числа (2, –5, π и т. д.). Считается, что для каждой константы однозначно задан предмет, который она обозначает. Далее для каждой константы четко указывается сорт, которому она принадлежит. Аналогией этого могут служить описания типизированных констант в языках программирования.

Столь же просты с виду и *переменные*, например, x, y,… Но для переменной неизвестен предмет, который она обозначает; в принципе она может обозначать какой угодно предмет из нашего универсума. Например, если наш универсум – люди, то x может обозначать в данный момент любого конкретного человека. Чтобы наши рассуждения не стали ошибочными, нужно следить, чтобы однажды выбранное значение x далее внутри данного рассуждения не изменялось, как говорят, оно должно быть *фиксированным* (обратите внимание, на противоположное понимание переменной в программировании). Для того

чтобы у нас был неограниченный запас имен переменных, часто пользуются индексами, например, x_1.

Более сложные выражения образуются применением символов операций к более простым. Операция, соответствующая символу, применяется к предметам и в результате дает тоже предмет. Например, символу × сопоставляется операция над числами, дающая по двум числам их произведение. В общем случае n-местную операцию f, примененную к выражениям $t_1,\ldots,t_n$, будем обозначать $f(t_1,\ldots,t_n)$, такую форму записи называют *функциональной*.

Выражение, обозначающее предмет, называется *термом*.

Операции называются еще *функциональными символами*, или просто *функциями*.

2.3.2 Предикаты и элементарные формулы

Чтобы образовать высказывание из предметов, нужно соединить их отношением; *n-местное отношение* – операция, сопоставляющая n предметам высказывание. Например, двуместное отношение «=» сопоставляет двум числам x и y высказывание «$x = y$», в частности 2 = 2 – истинное высказывание, а 2 = 5 – ложное высказывание. Одноместное отношение « … – положительное число» для числа 5 является истинным высказыванием «5 – положительное число». В «теории человеческих отношений» двуместное отношение «любить» сопоставляет паре (Ромео, Джульетта) истинное высказывание, а паре (Демон, Тамара) – ложное высказывание.

В математике чаще всего встречаются одноместные и двуместные (бинарные) отношения. Бинарные отношения обычно записываются между своими аргументами, например, $4 < 7$, $x^2 + 2x + 1 > 0$ и т. д. Одноместные отношения в математике часто записываются при помощи символа $\in$ и символа для множества объектов, обладающих данным свойством. Например, утверждение «π – действительное число» записывается в виде $\pi \in \boldsymbol{R}$, где $\boldsymbol{R}$ обозначает множество действительных чисел.

В логике для единообразия мы пользуемся записью

$$P(t_1,\ldots,t_n),$$

чтобы обозначить высказывание, образованное применением n-местного отношения P к предметам $t_1,\ldots,t_n$. Символ P, изображающий отношение называется *предикатом*. «Предикат» и «отношение» соотносятся как имя и предмет, им обозначаемый. Но в математике эти два понятия употребляются почти как синонимы. В логических материалах мы будем пользоваться строгим термином «предикат», а в конкретных приложениях, когда это вошло в математическую традицию, использовать и слово «отношение» (например, говорить об отношении «>» в формуле $a > b$).

В такой записи 2 = 4 выглядит следующим образом:

$$=(2, 4).$$

Элементарные (или *атомарные*) *формулы* имеют вид $P(t_1,\ldots,t_n)$, где P – n-местный предикат, $t_1,\ldots,t_n$ – термы. В обычной математике элементарные формулы называются просто *формулами*.

Сложные формулы строятся из элементарных. Задавая язык конкретной математической теории, непосредственно определяют именно элементарные формулы и их смысл.

Для того чтобы задать элементарные формулы, необходимо определить предикаты, используемые в нашей теории, и ее термы. А чтобы задать термы, нужно определить сорта объектов, константы и операции. В совокупности предикаты, сорта, константы, операции составляют *словарь* (или *сигнатуру*) теории.

Интерпретация

Задав словарь теории, необходимо *проинтерпретировать* все понятия, перечисленные в нем. При этом константам сопоставляются конкретные объекты, задаются правила вычисления функций, сопоставленных операциям, и правила, по которым определяются логиче-

ские значения предикатов. После интерпретации элементарные формулы, не содержащие переменных, оказываются либо истинны, либо ложны, а формулы, содержащие переменные, становятся истинными либо ложными после задания (фиксации) значений переменных.

Пример. В элементарной теории действительных чисел имеется единственный сорт объектов, интерпретируемый как множество действительных чисел **R**, двуместные отношения = и >, константы 0 и 1, операции +, –, × и /. Перечисленные символы составляют словарь этой теории. Но так работать невозможно, и в логике выработаны многочисленные способы вводить определения новых объектов и новых отношений таким образом, чтобы в принципе все выкладки и рассуждения, их использующие, можно было чисто механически расшифровать через исходные понятия.

Резюме

- Высказывания принимают логические значения.
- Единственными общепризнанными логическими значениями являются **истина** и **ложь**.
- Каждая теория имеет свой *универсум*, т. е. множество рассматриваемых предметов. Если универсумов несколько, то они называются *сортами* либо *типами*.
- Высказывания об предметах образуются при помощи *отношений*, или *предикатов*.
- Выражения, обозначающие предметы, называются *термами.*
- Термы строятся из переменных и констант при помощи операций, или *функциональных символов*, которые применяются к предметам и в результате дают предмет.
- Отношения (предикаты) применяются к термам и в результате дают высказывание (элементарную формулу).

2.3.3 Логические связки

Выражения с помощью которых записываются высказывания в нашем формальном языке называются *логическими формулами*. С чисто формальной точки зрения предикаты (отношения) можно рассматривать как функции, сопоставляющие своим аргументам истинностные значения, т. е. функции, принимающие всего два значения: **истина** и **ложь**.

Как только задана интерпретация и фиксированы значения всех встречающихся в элементарной формуле переменных, становится известно и логическое значение элементарной формулы.

Мы тогда можем определить истинностные значения и всех более сложных формул, так как значения их элементарных частей уже заданы. Но для этого нужно знать, какими способами более сложные формулы строятся из более простых.

Для образования новых формул из имеющихся используются логические связки. Логические связки применяются к высказываниям и в результате дают высказывания. Рассмотрим общепринятые логические связки. Через *A* и *B* будем обозначать произвольные высказывания.

Связка «и». Союзу «и» сопоставляется логическая связка &. Символ & называется конъюнкцией. Эта связка применяется при переводе на формальный язык утверждений вида:
«*A* и *B*»,
«*A*, но и *B* также»,
«*A* вместе с *B*»,
«*A*, несмотря на *B*»,
«не только *A*, но и *B*»,

«как *A*, так и *B*»,
«*A*, хотя и *B*»
и т. п. Все они переводятся одинаково: *A* & *B*. Разные слова здесь отражают разное отношение к факту, не меняя самого факта. Соответственно, переводя *A* & *B* на естественный язык, нужно выбирать подходящий, наиболее выразительный вариант.

Конъюнкция
Утверждение *A* & *B* истинно в том и только в том случае, когда истинны как *A*, так и *B*, и ложно в остальных случаях.

Связка «или». «*A* или *B*» символически записывается $A \vee B$. Знак $\vee$ называется *дизъюнкцией*. Эта же связка применяется при переводе утверждений
«*A* или *B* или оба вместе»,
«либо *A*, либо *B*»,
«*A* и/или *B*»
и т. п.

Дизъюнкция
Утверждение $A \vee B$ ложно в том и только в том случае, когда ложны как *A*, так и *B*, и истинно в остальных случаях.

Связка «следует». «Из *A* следует *B*» символически записывается $A \supset B$. Знак $\supset$ называется *импликацией*. Другими вариантами содержательных утверждений, точно также переводящихся, служат:
«*A* достаточное условие для *B*»,
«*B* необходимое условие для *A*»,
«*A*, только если *B*»,
«*B*, если *A*»,
«в случае *A* выполнено и *B*»,
«*A* есть *B*»,
«*A* влечет *B*».

Правила вычисления истинностного значения $A \supset B$ нуждаются в комментариях. Они опираются на содержательный смысл связки $\supset$: из *A* можно сделать вывод (вывести следствие) *B*, и на наши гипотезы (соглашения 1–3).

Рассмотрим верное утверждение: «Если *n* делится на 6, то *n* делится на 3». Получим, в частности, следующие три утверждения:

Если 6 делится на 6, то 6 делится на 3. 1
Если 5 делится на 6, то 5 делится на 3. 2
Если 3 делится на 6, то 3 делится на 3. 3

Все эти утверждения также обязаны быть истинными. Но, пользуясь соглашением 2 и заменяя утверждения о делимости на 6 на их конкретные логические значения, получаем, что тогда должно быть

(И $\supset$ И) = И
(Л $\supset$ Л) = И
(Л $\supset$ И) = И

Другими словами, должны быть истины утверждения

Из истины следует истина. 4
Из лжи следует ложь. 5
Из лжи следует истина. 6

Истинность 1-3 мы должны принять, если мы желаем обеспечит возможность подстановки в доказанные теоремы конкретных значений переменных. А по соглашению 2 нам приходится принять и 4-6.

Утверждение 3 и соответствующее ему утверждение 6 кажутся несколько парадоксальными. Но мы знаем, что из ложных предположений можно иногда содержательным рассуждением получить истинные следствия. Например, из ложного предположения «существуют русалки» следует истинное «купаться ночью в одиночку в незнакомом месте опасно». Принципиально неправильная система мира Птолемея, в которой центром Вселенной служит Земля, очень точно описывает видимые движения планет. Соглашение 2 опять–таки заставляет нас распространить эту истинность на все мыслимые в математике случаи.

Правда, при этом приходится признать формально истинными и предложения типа

«Если 2×2 = 5, то снег черный».

Импликация

Утверждение $A \supset B$ ложно в том и только в том случае, когда A истинно и B ложно, и истинно во всех остальных случаях.

Связка «тогда и только тогда». «A тогда и только тогда, когда B» символически записывается $A \sim B$. Знак ~ называется *эквивалентность* (или *эквиваленция*). Той же связкой переводятся предложения:
«A эквивалентно B»,
«A необходимое и достаточное условие для B»,
«если A, то B и наоборот»
и т. п.

Эквивалентность

Утверждение $A \sim B$ истинно тогда и только тогда, когда истинностные значения A и B совпадают, и ложно в противном случае.

Очевидно, можно считать, что $A \sim B$ есть сокращенная запись формулы $(A \supset B)\ \&\ (B \supset A)$.

Связка «не». Утверждение «не A» символически записывается $\neg A$. Знак $\neg$ называется *отрицанием*. Эта же связка используется при переводе выражений:
«A неверно»,
«A ложно»,
«A не может быть»
и т. п.

Отрицание

Утверждение $\neg A$ истинно тогда и только тогда, когда A ложно, и ложно в противном случае.

Связки $\&$, $\vee$, $\supset$, $\sim$ и $\neg$ называются *связками исчисления высказы*ваний или *пропозициональными связками.*

Квантор «для всех». Утверждение «для всех x верно $A(x)$» символически записывается $\forall x\ A(x)$. Символ $\forall$ называется *квантором всеобщности* (или *универсальным квантором*). Эта же связка используется при переводе утверждений:
«A верно при любом значении x»,
«для произвольного x имеет место $A(x)$»,

«каково бы ни было x, $A(x)$»,
«для каждого x (верно) $A(x)$»,
«всегда имеет место $A(x)$»,
«каждый обладает свойством A»,
«свойство A присуще всем»
и т.п. Утверждение $\forall x\ A(x)$ истинно тогда и только тогда, когда $A(c)$ истинно, какой бы конкретный предмет c из универсума нашей теории мы ни подставляли вместо x.

Квантор общности

Утверждение $\forall x\ A(x)$ истинно тогда и только тогда, когда $A(x)$ истинно при любом фиксированном значении x. Утверждение $\forall x\ A(x)$ ложно тогда и только тогда, когда имеется хоть один предмет c из нашего универсума (другими словами, хотя бы одно значение x), такой, что $A(c)$ ложно.

Отрицание

Утверждение $\neg A$ истинно тогда и только тогда, когда A ложно, и ложно в противном случае.

Связки $\&$, $\vee$, $\supset$, $\sim$ и $\neg$ называются *связками исчисления высказы*ваний или *пропозициональными связками.*

Квантор «для всех». Утверждение «для всех x верно $A(x)$» символически записывается $\forall x\ A(x)$. Символ $\forall$ называется *квантором всеобщности* (или *универсальным квантором*). Эта же связка используется при переводе утверждений:
«A верно при любом значении x»,
«для произвольного x имеет место $A(x)$»,
«каково бы ни было x, $A(x)$»,
«для каждого x (верно) $A(x)$»,
«всегда имеет место $A(x)$»,
«каждый обладает свойством A»,
«свойство A присуще всем»
и т.п. Утверждение $\forall x\ A(x)$ истинно тогда и только тогда, когда $A(c)$ истинно, какой бы конкретный предмет c из универсума нашей теории мы ни подставляли вместо x.

Квантор общности

Утверждение $\forall x\ A(x)$ истинно тогда и только тогда, когда $A(x)$ истинно при любом фиксированном значении x. Утверждение $\forall x\ A(x)$ ложно тогда и только тогда, когда имеется хоть один предмет c из нашего универсума (другими словами, хотя бы одно значение x), такой, что $A(c)$ ложно.

Выразительные средства языка, который мы описываем, принципиально ограничены в одном важном отношении: нет возможности говорить о произвольных свойствах объектов теории, т. е. о произвольных подмножествах множества всех объектов. Синтаксически это отражается в запрете формулировать выражения, скажем вида $\forall P(P(x))$, где P – предикат. Предикаты обозначают фиксированные, а не переменные свойства. Поэтому данный язык называется *языком логики предикатов первого порядка* (или просто *языком первого порядка*).

Фрагмент языка первого порядка, где элементарные формулы обозначаются просто переменными (называемыми «высказывательными» или «пропозициональными»), а кванторы совсем не используются называется языком логики высказываний.

Языки, в которых допускаются кванторы по свойствам (предикатам) и (или) по функциям (а также, возможно, по свойствам свойств и т. д.), называются языками высших

порядков. В таком языке мы можем также ввести и какие-то новые кванторы, а не только кванторы общности и существования.

Резюме

- Формулы – выражения, обозначающие высказывания в языке логики предикатов.
- Сложные формулы строятся из более простых при помощи *логических связок*.
- Логические связки применяются к высказываниям и в результате дают высказывание.
- Логические связки делятся на связки исчисления высказываний (или пропозициональные) и кванторы.

2.3.4 Свободные и связанные переменные

Обсудим очень важные понятия свободной и связанной переменных. В античной математике (формальные) переменные практически не использовались. Под влиянием Виета в конце 16 столетия переменные стали стандартным инструментом математики. Было, однако, много недоразумений по поводу природы переменных. Например, целочисленная переменная была «изменяющейся величиной», которая иногда четна, иногда нечетна и может быть «зафиксирована» на время доказательства. Такие сущности, однако, не существуют.

Фреге и Пирс прояснили истинную природу переменных: это – синтаксические объекты, и следует различать свободные и связанные переменные.

Свободная переменная – это синтаксический объект, встречающийся в некотором контексте, вместо которого можно подставлять другие синтаксические объекты. Когда терм, содержащий свободные переменные, интерпретируется в некоторой универсуме M, нужно выбрать элементы предметной области M, чтобы интерпретировать эти переменные.

С другой стороны, связанные переменные не допускают ни подстановки, ни выбора интерпретации. В следующих примерах переменная x связанная:

$$\int_0^1 x^2 + 1\, dx; \quad \sum_{x=0}^{100} x^2 + 1.$$

Если терм t подставляется (без предосторожностей) вместо свободных вхождений переменной y в выражение $A \equiv A(y)$, то некоторые переменные в t могут стать связанными. Пусть $A(y)$ – истинная формула

$$\int_0^1 (x^2 + y)\, dx = y + \tfrac{1}{3},$$

и пусть $t \equiv x$; тогда $A(x)$ – ложная формула

$$\int_0^1 (x^2 + x)\, dx = x + \tfrac{1}{3}.$$

Это явление называется *коллизией переменных*.

Говорят, что подстановка терма t вместо y в A корректна, если никакая свободная переменная терма t не становится связанной после подстановки t вместо (свободных вхождений) переменной y в A. Именно некорректная подстановка вызвала только что упомянутую коллизию переменных. Переименовывая связанные переменные в A, мы всегда можем избежать коллизии переменных. Например, подстановка $t \equiv x$ вместо y в формулу

$$\int_0^1 (u^2 + y)\, du = y + \tfrac{1}{3}$$

является корректной.

Введение кванторов общности и существования приводит снова к противопоставлению свободных и связных вхождений переменной в термы и формулы.

Свободные и связные вхождения переменных

а) Всякое вхождение переменной в элементарную формулу или терм свободно.

б) Всякое вхождение переменной в $\neg P$ или в $P \square Q$ ($\square$ – любая пропозициональная связка) свободно (соответственно связанно) в точности тогда, когда свободно (соответственно связано) соответствующее вхождение в P или Q.

в) Всякое вхождение переменной x в $\forall xP$ и $\exists xP$ связано. Вхождение остальных переменных в $\forall xP$ и $\exists xP$ таковы же, как соответствующие вхождения в P.

Пусть дано вхождение квантора $\forall$ (или $\exists$) в формулу P. Из определений следует, что вслед за ним в P входит переменная и некоторая подформула Q (которая является либо элементарной формулой либо начинается с открывающей скобкой). Выражение xQ, начинающее с этой переменной и кончающееся соответствующей закрывающей скобкой, называется *областью действия* данного (вхождения) квантора.

Теперь мы можем ввести важный класс *замкнутых* формул. По определению, это – формулы без свободных переменных. Интуитивный смысл понятия замкнутой формулы таков. Оно отвечает вполне определенному (в частности, в отношении истинности или ложности) высказыванию: имена неопределенных объектов теории используются только в контексте «все объекты x удовлетворяют условию ...» или «существует объект x со свойством ...». Наоборот, незамкнутая формула $A(x)$ или $\exists x\, A(x,y)$ может быть истинной или ложной в зависимости от того, какие предметы нарекаются именами x (для первой); y (для второй). Истинность или ложность понимаются здесь для фиксированной интерпретации языка.

2.4 Теории с языком первого порядка

2.4.1 Интерпретация

Формулы теории имеют смысл только тогда, когда имеется какая-нибудь интерпретация входящих в них символов.

Если $\boldsymbol{T}$ – теория с языком первого порядка, то можно рассматривать различные интерпретации этой теории. Чтобы определить *интерпретацию* мы должны задать:

- универсум M, называемый *областью интерпретации*;
- соответствие, относящее
    - каждому n-местному предикату некоторое n-местное отношение в M,
    - каждой функциональной букве, требующей n аргументов, – некоторую функцию $M^n \to M$ и
    - каждой константе – некоторый элемент из M;
- предметные переменные мыслятся пробегающими область интерпретации M;
- логическим связкам придаются их обычный смысл.

Для заданной интерпретации всякая формула P без свободных переменных представляет собой высказывание, которое истинно или ложно. Если высказывание является истинным, то говорят, что формула P *выполняется* в данной интерпретации.

Всякая формула со свободными переменными выражает некоторое отношение на области интерпретации; это отношение может быть выполнено (истинно) для одних значений переменных из области интерпретации и не выполнено (ложно) для других.

Формализация задается не только синтаксисом и семантикой формального языка (эти компоненты как раз чаще всего берутся традиционными, из хорошо известного крайне ограниченного набора), но и множеством утверждений, которые считаются истинными. Именно эта формулировка базисных свойств, аксиом, описывающих некоторую предмет-

ную область, обычно рассматривается как математическое описание объектов. Таким образом, практически нас интересует не все интерпретации данной теории, а лишь те из них на которых выполнены аксиомы.

Модели

- Интерпретация называется *моделью множества формул* Г, если все формулы этого множества выполняются в данной интерпретации.
- *Моделью теории* называется такая интерпретация, в которой истинны все теоремы теории.

2.4.2 Общезначимость и непротиворечивость

- Формула P называется **общезначимой**, если она истинна в каждой интерпретации (обозначается $\models P$).
- Формула P называется **противоречием**, если формула P ложна во всякой интерпретации.
- Формула P называется **логическим следствием** множества формул Г, если P выполняется в любой модели Г.
- Формула B является *логическим следствием* формулы A (обозначение: $A \Rightarrow B$), если формула B выполнена в любой интерпретации, в которой выполнена формула A.
- Формулы A и B *логически эквивалентны* (обозначение: $A=B$), если они являются логическим следствием друг друга.
- Формальная теория $\boldsymbol{T}$ называется **семантически непротиворечивой**, если ни одна ее теорема не является противоречием.

Формальная теория пригодна для описания тех предметных областей, которые являются ею моделями.

Справедлива метатеорема:

Модель для формальной теории $\boldsymbol{T}$ существует тогда и только тогда, когда $\boldsymbol{T}$ семантически непротиворечива.

Обсудим подход к разрешению противоречий, принятый в математике [24]. Формальные системы, использующие логический вывод, не могут содержать противоречий – противоречия мгновенно заражают всю систему, подобно глобальному раку. Это не похоже на человеческую мысль. Если вы обнаружите в своих рассуждениях противоречие, вряд ли это разрушит все здание вашего мышления. Вместо этого вы, скорее всего, попытаетесь найти те идеи или методы рассуждения, которые явились причиной противоречия. Иными словами, вы попытаетесь, насколько это возможно, выйти из ваших внутренних систем, приведших к противоречию, и попробуете их исправить. Маловероятно, что вы поднимите руки вверх и воскликнете: «Это показывает, что теперь я верю во все, что угодно!» – разве что в шутку.

В действительности, противоречия – это важный источник прогресса во всех областях жизни, и математика не является исключением. В прошлом, когда в математике обнаруживалось противоречие, математики тут же пытались найти виновную в этом систему, выйти из таковой, проанализировать её и, если возможно, залатать прореху. Вместо того, чтобы делать математику слабее, нахождение и «починка» противоречивых систем только усиливали её.

2.4.3 Полнота, независимость и разрешимость

Пусть универсум M, рассматриваемый с соответствующей интерпретацией, является моделью формальной теории $\boldsymbol{T}$.

- Формальная теория ***T*** называется ***полной*** (относительно данной интерпретации), если каждому истинному высказыванию об объектах *M* соответствует теорема теории ***T*** .
- Если для предметной области *M* существует формальная полная непротиворечивая теория ***T***, то *M* называется **аксиоматизируемой** (или **формализуемой**).
- Система аксиом (или аксиоматизация) формально непротиворечивой теории ***T*** называется **независимой**, если никакая из аксиом не выводима из остальных по правилам вывода теории ***T***.

Одним из первых вопросов, которые возникают при задании формальной теории, является вопрос о том, возможно ли, рассматривая какую-нибудь формулу формальной теории, определить, является ли она доказуемой или нет. Другими словами, речь идет о том, чтобы определить, является ли данная формула теоремой или *не-теоремой* и как это доказать.

- Формальная теория ***T*** называется **разрешимой**, если существует алгоритм, который для любой формулы теории определяет, является ли это формула теоремой теории.
- Формальная теория ***T*** называется **полуразрешимой**, если существует алгоритм, который для любой формулы *P* теории выдает ответ «да», если *P* является теоремой теории и выдает «нет» или, может быть, не выдает никакого ответа, если *P* не является теоремой (то есть алгоритм применим не ко всем формулам).

2.5 Примеры формальных теорий

Приведем несколько примеров формальных теорий из книги Хофштадтера [9].

Формальная система *MIU*

Алфавит: *M, I, U*.
Формулы = {*M, I, U*}*.
Аксиома: *MI*.
Правила вывода:

1). $xI \to xIU$ (продукция);
2). $Mx \to Mxx$ (продукция);
3). $III \to U$ (правило переписывания);
4). $UU \to \varnothing$ (правило переписывания, $\varnothing$ обозначает пустую последовательность).

- Продукция – правило, применяемое к формулам, рассматриваемым как единое целое.
- Правило переписывания – правило, которое может применяться к любой подформуле формулы.

Приведем типичный вывод в этой теории:

MI	Аксиома
MII	Правило 2
MIIII	Правило 2
MUI	Правило 3
MUIU	Правило 1
MUIUUIU	Правило 2
MUIIU	Правило 4

Любые утверждения о свойствах этой теории являются метатеоремами. Предлагается читателю задача: «Найдите вывод *MU* или докажите, что он невозможен».

Формальная система PR

Алфавит: {P, R, –}
Выражения – элементы {P, R, –}*.

Формулы – строки (последовательности) вида xPyRz, где x, y и z – строчки, состоящие только из тире.
Схема аксиом:
xP–Rx– является аксиомой, когда x состоит только из тире (каждое из двух вхождений x замещает одинаковое число тире).
Правило вывода (схема):
Пусть x, y и z – строчки, состоящие только из тире. Пусть xPyRz является теоремой. Тогда xPy–Rz– также будет теоремой.

В системе PR используются только удлиняющие правила, т. е. количество символов в формуле в результате применения правила вывода увеличивается.

Задачи. 1. Докажите: *Формальная система, имеющая только удлиняющие правила (и не имеющая укорачивающих правил) имеет разрешающий алгоритм*.
2. Найдите разрешающий алгоритм для теорем *PR*.

Прежде, чем читать дальше попробуйте решить данные задачи.

Разрешающий алгоритм для формальных систем, имеющих только удлиняющие правила, можно сформулировать в следующем виде.

Пусть F – проверяемая формула и M – множество всех аксиом теории.

While $\forall x \in M((\text{длина}(x) \le \text{длина}(F))\ \&\ F \notin M$ **do**

Begin

Для каждой формулы $x \in M$ выполняем следующее:

Begin

Пусть R_x обозначает множество всех формул, получаемых из x однократным применением правил вывода.

Полагаем M равным $(M \cup R_x) \setminus \{x\}$.

End

End

If $F \in M$ **then** F – теорема **else** F – не теорема.

Выберем следующую интерпретацию системы *PR* (одна из возможных).

- Универсум – множество целых положительных чисел.
- Последовательность, состоящая из n тире, интерпретируется как число n.
- P интерпретируется как символ +.
- R интерпретируется как символ =.

Нетрудно убедиться, что указанная интерпретация теореме *xPyRz* ставит в соответствие истинное утверждение о целых положительных числах «$x+y=z$» и поэтому данная интерпретация является *моделью* системы *PR*.

Теперь мы можем использовать более простой разрешающий алгоритм для теории *PR*: формула *xPyRz* является теоремой тогда и только тогда, когда $x+y=z$ – истина.

В модели теоремы и истины совпадают – то есть, между теоремами и фрагментами реального мира существует изоморфизм.

Грубо говоря: «изоморфизм» есть преобразование, сохраняющее информацию. Слово «изоморфизм» приложимо к тем случаям, когда две сложные структуры могут быть отображены одна в другую таким образом, что каждой части одной структуры соответствует какая-то часть другой структуры («соответствие» здесь означает, что эти части выполняют в своих структурах сходные функции).

При данной интерпретации есть изоморфизм между системой *PR* и сложением.

Формальная система *UR*

Алфавит: $\{U, R, -\}$
Выражения – элементы $\{U, R, -\}^*$.
Формулы – строки вида *xUyRz*, где x, y и z – строчки, состоящие только из тире.
Схема аксиом:

xU–Rx является аксиомой, когда *x* состоит только из тире (каждое из двух вхождений *x* замещает одинаковое число тире).
Правило вывода (схема):
Пусть *x*, *y* и *z* – строчки, состоящие только из тире. Пусть *xUyRz* является теоремой. Тогда *xUy–Rzx* также будет теоремой.

Задача. Найдите разрешающий алгоритм для теорем *UR*.

Выберем следующую интерпретацию системы *UR*.

- Универсум – множество целых положительных чисел.
- Последовательность, состоящая из *n* тире, интерпретируется как число *n*.
- *P* интерпретируется как символ ×.
- *R* интерпретируется как символ =.

Нетрудно убедиться, что указанная интерпретация теореме *xUyRz* ставит в соответствие истинное утверждение о целых положительных числах «*x*×*y*=*z*» и поэтому данная интерпретация является *моделью* системы *UR*.

Формальная система *PR*1

Алфавит: {*P*, *R*, –}
Выражения – элементы {*P*, *R*, –}*.
Формулы – строки вида *xPyRz*, где *x*, *y* и *z* – строчки, состоящие только из тире.
Схемы аксиом:

1. *xP–Rx–* является аксиомой, когда *x* состоит только из тире (каждое из двух вхождений *x* замещает одинаковое число тире).
2. *xP–Rx* является аксиомой, когда *x* состоит только из тире (каждое из двух вхождений *x* замещает одинаковое число тире).

Правило вывода (схема):
Пусть *x*, *y* и *z* – строчки, состоящие только из тире. Пусть *xPyRz* является теоремой. Тогда *xPy–Rz–* также будет теоремой.

Задача. Найдите разрешающий алгоритм для теорем *PR*1.

Рассмотрим различные интерпретации системы *PR*1.

1. Выберем интерпретацию системы *PR*1 такую же, как для *PR*.

- Универсум – множество целых положительных чисел.
- Последовательность, состоящая из *n* тире, интерпретируется как число *n*.
- *P* интерпретируется как символ +.
- *R* интерпретируется как символ =.

Указанная интерпретация теореме *xPyRz* ставит в соответствие утверждение о целых положительных числах «*x*+*y*=*z*». Но эти утверждения могут быть и ложными, поэтому данная интерпретация не является *моделью* системы *PR*1.

2. Вторая интерпретация системы *PR*1 отличается от первой только тем, как интерпретируется символ *R*.

- *R* интерпретируется как «равняется или больше на 1».

Указанная интерпретация теореме *xPyRz* ставит в соответствие истинное утверждение о целых положительных числах «*x*+*y*=*z*+1 ∨ *x*+*y*=*z*» и поэтому данная интерпретация является *моделью* системы *PR*1. Более того, любое истинное утверждение «*x*+*y*=*z*+1 ∨ *x*+*y*=*z*» описывается в виде теоремы *xPyRz* теории *PR*1.

3. В последней интерпретации символ *R* понимается снова по-другому.

- *R* интерпретируется как символ ≥».

Указанная интерпретация теореме *xPyRz* ставит в соответствие истинное утверждение о целых положительных числах «*x*+*y*≥*z*»и поэтому данная интерпретация является *моделью* системы *PR*1. Но мы сейчас имеем существенное отличие от предыдущей интерпретации: не все истинные утверждения вида *x*+*y*≥*z* являются теоремами в *PR*1. Так, например, формула --*P*-*R*- имеет истинную интерпретацию 2+1 ≥ 1, но это не теорема.

3 Исчисление высказываний

3.1 Определение исчисления высказываний

Исчислением высказываний называется формальная теория с языком логики высказываний, со схемами аксиом

$A1)$ $A \supset (B \supset A)$;

$A2)$ $(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$;

$A3)$ $(\neg B \supset \neg A) \supset ((\neg B \supset A) \supset B)$;

и правилом вывода *MP* (*Modus Ponens* – обычно переводится как *правило отделения*):

$$\frac{A,\ A \supset B}{B}\ MP$$

Здесь A, B и C – *любые* формулы. Таким образом, множество аксиом исчисления высказываний бесконечно, хотя задано тремя схемами аксиом. Множество правил вывода также бесконечно, хотя оно задано только одной схемой.

Пример. Для любой формулы A построим вывод формулы $A \supset A$, т. е. $A \supset A$ – теорема.

Подставляем в схему аксиом $A2$ вместо B формулу $A \supset A$ и вместо C формулу A, получаем аксиому

$$(A \supset ((A \supset A) \supset A)) \supset ((A \supset (A \supset A)) \supset (A \supset A)). \quad (1)$$

Подставляем в $A1$ вместо формулы B формулу $A \supset A$, получаем аксиому

$$A \supset ((A \supset A) \supset A). \quad (2)$$

Из формул (1) и (2) по правилу *MP* получаем

$$(A \supset (A \supset A)) \supset (A \supset A). \quad (3)$$

Подставляем в $A1$ вместо формулы B формулу A, получаем аксиому

$$A \supset (A \supset A). \quad (4)$$

Из формул (3) и (4) по правилу MP получаем $A \supset A$.

В исчислении высказываний импликация очень тесно связана с выводимостью.

Теорема 1 (о дедукции). Если $\Gamma, A \vdash B$, то $\Gamma \vdash A \supset B$ и наоборот.

Теорема 2. (Пост, 1921) Формула A в исчислении высказываний является теоремой тогда и только тогда, когда A – тавтология.

3.2 Разрешимость и непротиворечивость исчисления высказываний

Интерпретация формул исчисления высказываний проста – область интерпретации состоит из двух значений «истина» и «ложь»; поэтому пропозициональная переменная принимает только значения И и Л и интерпретация составной формулы вычисляется по известным законам с помощью логических операций над истинностными значениями. Поскольку любая формула содержит только конечное число пропозициональных переменных, то формула обладает только конечным числом различных интерпретаций. Следовательно, исчисление высказываний является, очевидно, разрешимой формальной теорией.

Легко убедиться, что исчисление высказываний является формально непротиворечивой теорией. Действительно, все теоремы исчисления высказываний суть тавтологии. Отрицание тавтологии не есть тавтология. Следовательно, никакая формула не может быть выведена вместе со своим отрицанием.

4 Теории первого порядка

4.1 Синтаксические свойства истинности теорий с языком первого порядка

Нам понадобится следующее понятие языка первого порядка. Пусть дана формула P, свободное вхождение переменной x в P и терм t. Мы говорим, что *данное вхождение x не связывает t в P*, если оно не лежит в области действия ни одного квантора вида $\forall y$ и $\exists y$, где y - переменная, входящая в t.

Иными словами, после подстановки t вместо данного вхождения x все переменные, входящие в t, останутся свободными в P.

Чаще всего приходится подставлять терм вместо каждого из свободных вхождений данной переменной. Важно, что такая операция переводит термы в термы и формулы в формулы. Если каждое свободное вхождение x в P не связывает t, мы будем говорить просто, что *терм t свободный для x в P*.

Пример

1. Терм y свободен для переменной x в формуле $P(x)$, но тот же терм y не свободен для переменной x в формуле $\forall y\, P(x)$.
2. Терм $f(x,z)$ свободен для переменной x в формуле $\forall y P(x, y) \supset Q(x)$, но тот же терм $f(x,z)$ не свободен для переменной x в формуле $\exists z \forall y\, P(x, y) \supset Q(x)$.

Пусть нам даны некоторая формальная теория $\boldsymbol{T}$ с языком первого порядка и задана интерпретация этой теории. Обозначим через $\boldsymbol{F}$ множество всех формул теории $\boldsymbol{T}$, истинных в данной интерпретации. Перечислим те свойства $\boldsymbol{F}$, которые отражают заложенные в языки первого порядка логику, независящую от конкретных особенностей интерпретации.

- Для любой замкнутой формулы P, либо $P \in \boldsymbol{F}$, либо $\neg P \in \boldsymbol{F}$.
- Множество $\boldsymbol{F}$ не содержит противоречия, т. е. ни для какой формулы P не может быть, чтобы одновременно выполнялось $P \in \boldsymbol{F}$ и $\neg P \in \boldsymbol{F}$.
- Множество $\boldsymbol{F}$ содержит все тавтологии.
- Множество $\boldsymbol{F}$ содержит следующие общезначимые формулы (называемые «логические аксиомы с кванторами»):
  $\forall x\, A(x) \supset A(t)$,
  где $A(t)$ есть формула теории $\boldsymbol{T}$ и t есть терм теории $\boldsymbol{T}$, свободный для x в $A(x)$. Заметим, что t может совпадать с x, и тогда мы получаем аксиому $\forall x A(x) \supset A(x)$.
  $\forall x\, (A \supset B(x)) \supset (A \supset \forall x\, B(x))$,
  где A не содержит свободных вхождений переменной x.
- Множество $\boldsymbol{F}$ замкнуто относительно правил вывода modus ponens и обобщения. По определению это означает, что если $A \in \boldsymbol{F}$ и $A \supset B \in \boldsymbol{F}$, то также $B \in \boldsymbol{F}$; если $A \in \boldsymbol{F}$, то $\forall x A \in \boldsymbol{F}$ для любой переменной x.

4.2 Определение теории первого порядка

Аксиоматическая формальная теория с языком первого порядка называется *теорией первого порядка*, если она имеет следущие аксиомы и правила вывода..

Аксиомы теории первого порядка $\boldsymbol{T}$ разбиваются на два класса: логические аксиомы и собственные (или нелогические) аксиомы.

Логические аксиомы: каковы бы ни были формулы A, B и C теории $\boldsymbol{T}$, следующие формулы являются логическими аксиомами теории $\boldsymbol{T}$.

$\boldsymbol{A_1}$. $A \supset (B \supset A)$;

$\boldsymbol{A_2}$. $(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$;

$\boldsymbol{A_3}$. $(\neg B \supset \neg A) \supset ((\neg B \supset A) \supset B)$;

$\boldsymbol{A_4}$. $\forall x\, A(x) \supset A(t)$, где $A(t)$ есть формула теории $\boldsymbol{T}$ и t есть терм теории $\boldsymbol{T}$, свободный для x в $A(x)$. Заметим, что t может совпадать с x, и тогда мы получаем аксиому $\forall x A(x) \supset A(x)$.

$\boldsymbol{A_5}$. $\forall x\,(A \supset B(x)) \supset (A \supset \forall x\, B(x))$, где A не содержит свободных вхождений переменной x.

Собственные аксиомы: таковые не могут быть сформулированы в общем случае, ибо меняются от теории к теории. Правилами вывода во всякой теории первого порядка являются

1. *Modus ponens*: $\dfrac{A,\ A \supset B}{B}$ *MP*
2. Правило обобщения: $\dfrac{A(x)}{\forall x A(x)}$ *Gen*

Теория первого порядка, которая не содержит собственных аксиом называется *исчислением предикатов первого порядка*. *Чистым исчислением предикатов* называется исчисление предикатов первого порядка, не содержащее предметных констант и функторов.

Как мы увидим позже, логические аксиомы выбраны таким образом, что множество логических следствий аксиом теории в точности совпадает с множеством теорем теории. В частности, для исчисления предикатов первого порядка множество его теорем совпадает с множеством логически общезначимых формул.

Общезначимые формулы в теории первого порядка играют ту же роль, что тавтологии в логике высказываний. Между ними есть и формальная связь: если взять любую тавтологию и вместо входящих в нее пропозициональных переменных подставить произвольные формулы языка первого порядка, то получится общезначимая формула. В самом деле, пусть есть некоторая интерпретация теории, при которой как-то зафиксированы значения предметных переменных. Тогда каждая из подставленных формул станет истинной или ложной, а значение всей формулы определяется с помощью таблиц истинности для логических связок, то есть по тем же правилам, что и в логике высказываний.

Конечно, бывают и другие общезначимые формулы, не являющиеся частным случаем пропозициональных тавтологий. Например, формула

$$\forall x A(x) \supset \exists y A(y)$$

общезначима (здесь существенно, что универсум любой интерпретации непуст).

4.3 Некоторые свойства теорий первого порядка

Для теорий первого порядка можно уточнить некоторые понятия, введенные ранее.

Формула P является **противоречием,** тогда и только тогда, когда формула $\neg P$ является общезначимой.

Формальная теория $\boldsymbol{T}$ называется (**формально**) **непротиворечивой**, если в ней не являются выводимыми одновременно формулы P и $\neg P$.

Имеет место метатеорема:

Теория $\boldsymbol{T}$ формально непротиворечива тогда и только тогда, когда она семантически непротиворечива.

Из определений логического следования и эквивалентности непосредственно вытекают следующие утверждения:

$A \Rightarrow B$ тогда и только тогда, когда $\models A \supset B$;

$A = B$ тогда и только тогда, когда $\models A \sim B$.

Имеют место также следующие логические следования и эквивалентности (формулы A и B – произвольны, формула C не содержит никаких вхождений переменной x):

$\neg\forall x A(x) = \exists x \neg A(x)$, $\quad \neg\exists x A(x) = \forall x \neg A(x)$;

$\forall x(A(x)\&B(x)) = \forall x\, A(x)\&\forall x B(x)$, $\quad \exists x(A(x)\vee B(x)) = \exists x A(x)\vee\exists x B(x)$;

$\exists x(A(x)\&B(x)) \Rightarrow \exists x A(x)\&\exists x B(x)$, $\quad \forall x A(x)\vee\forall x B(x) \Rightarrow \forall x(A(x)\vee B(x))$;

$\forall x\forall y A(x,y) = \forall y\forall x A(x,y)$, $\quad \exists x\exists y A(x,y) = \exists y\exists x A(x,y)$;

$\forall x(A(x)\&C) = \forall x A(x)\&C$, $\quad \forall x(A(x)\vee C) = \forall x A(x)\vee C$;

$\exists x(A(x)\&C) = \exists xA(x)\&C$, $\exists x(A(x)\vee C) = \exists xA(x)\vee C$;
$C\supset\forall xA(x) = \forall x(C\supset A(x))$, $C\supset\exists xA(x) = \exists x(C\supset A(x))$;
$\forall xA(x)\supset C \Rightarrow \exists x(A(x)\supset C)$.

Имеет место следующее утверждение:

Если теория первого порядка (формально) противоречива, то в ней выводима любая формула.

Доказательство. В самом деле, пусть формулы A и $\neg A$ выводимы в теории. Формула $\neg A\supset(A\supset B)$ является тавтологией в исчислении высказываний, следовательно, она выводима. Её вывод, поскольку он содержит только *MP*, остается выводом и в любой теории первого порядка. Поэтому формула $\neg A\supset(A\supset B)$ выводима в теории первого порядка. Дважды применяя *MP*, мы получаем вывод произвольной формулы B.

Таким образом, для доказательства непротиворечивости какой-либо теории первого порядка достаточно установить невыводимость в этой теории хотя бы одной формулы.

Теорема 3. (Гёдель, 1930). Всякая общезначимая формула теории ***T*** первого порядка является теоремой теории ***T***.

Значительная часть теорем логики состоит в доказательстве утверждений вида «Г|–*P*» или «не Г|–*P*» для разных теорий первого порядка, множеств Г и разных (классов) формул *P*.

Результат «Г|–*P*» может доказываться посредством предъявления описания вывода формулы *P* из Г. Однако в мало-мальски сложных случаях оно оказывается настолько длинным, что заменяется инструкцией по составлению такого описания, более или менее полной. Наконец, доказательство «Г|–*P*» может вообще не сопровождаться предъявлением вывода *P* из Г, хотя бы и неполного. В этом случае мы «не доказываем *P*, а доказываем, что существует доказательство *P*».

Результат «не Г|–*P*» в редких случаях может устанавливаться чисто синтаксическим рассуждением, но обычно доказательство опирается на конструкцию модели, т.е. интерпретации, в которой Г истинно, а *P* ложно.

4.4 Непротиворечивость, полнота и неразрешимость исчисления предикатов

Теорема 4. Исчисление предикатов первого порядка непротиворечиво.

Теорема 5. (о полноте исчисления предикатов). В исчислении предикатов первого порядка теоремами являются те и только те формулы, которые общезначимы.

Теорема 6. (Теорема Чёрча о неразрешимости исчисления предикатов, 1936). Не существует алгоритма, который для любой формулы исчисления предикатов первого порядка устанавливает, общезначима она или нет.

Теорема Чёрча остается в силе и для любой теории первого порядка, содержащей все формулы исчисления предикатов первого порядка.

Теорема 5 показывает, что в логике предикатов синтаксический метод теорий первого порядка равносилен семантическому методу, использующему понятие интерпретации, модели, общезначимости и т. п. Для исчисления высказываний имеет место аналогичная эквивалентность семантических (тавтология и др.) и синтаксических (теорема и др.) понятий. Отметим также, что для исчисления высказываний теорема о полноте системы приводит к решению проблемы разрешения. Однако для теорий первого порядка мы не можем получить разрешающий алгоритм для общезначимости или, что тоже самое, для выводимости в любом исчислении предикатов первого порядка.

5 Классические методы доказательства

Центральным понятием, введенным математикой, явилось доказательство. Само его появление коренным образом видоизменило стиль мышления значительной части людей и положило начало тому, что сейчас называют *рациональным научным мышлением*. Будем придерживаться содержательного определения доказательства, данного русским математиком Н. Н. Непейвода:

Доказательство – конструкция, синтаксическая правильность которой гарантирует семантическую.

Естественно, что столь общее и гибкое понятие, как доказательство, должно быть приложимо самыми разными способами. Выделим два вида использование доказательства.

Использование доказательства

1. Сведение новой задачи к уже решенным задачам.
2. Выявление условий, при которых можно пользоваться данным утверждением.

Чистые математики занимаются тем, что решают задачи. Никакое математическое доказательство не ведется с самого начала (за исключением нескольких примитивных теорем в учебниках логики и алгебры). Оно заканчивается ссылками на уже известные теоремы, которые когда-то были математическими задачами. Таким образом, как говорят в математическом фольклоре, «решить задачу – значит свести ее к уже решенным».

Эта привычка сводить новые задачи к уже решенным послужила даже основанием для шутки, которая на самом деле достаточно точно отражает суть математического метода:

Математику задали вопрос:

– Как приготовить чай?

– Элементарно. Берем чайник, наливаем в него воду, зажигаем газ, ставим чайник на огонь, ждем пока закипит, выключаем газ, кладем заварку в соответствующий сосуд, заливаем ее кипятком, ждем еще пять минут, и чай готов.

– А если у нас уже есть чайник с кипятком?

– Выливаем из него кипяток и сводим задачу к предыдущей.

Именно так и действует хороший математик, решая задачу.

Даже работая в рамках какой-нибудь формальной теории, математик пытается неформально подходить к доказательству, облегчая себе жизнь. Для этого уже доказанные теоремы используются наравне с аксиомами, кроме того используются новые правила вывода, обоснованность применения которых, следует из существующих правил вывода и аксиом.

Например, справедливо *цепное правило вывода*, которое позволяет вывести новую импликацию из двух данных импликаций. Можно записать его следующим образом:

$A \supset B, B \supset C \ |\!- A \supset C$.

Во всей своей полноте понятие доказательства несомненно обладает и психологическими признаками. Надо обладать красноречием и умением убеждать, чтобы слушатели (или читатели) приняли ваше доказательство. С неформальной точки зрения, *доказательство – это просто рассуждение, убеждающее нас настолько, что с его помощью мы готовы убеждать других.*

В классической логике используются следующие методы доказательства:

Классические методы доказательства

1. Использование теоремы о дедукции.
2. Доказательство импликаций с помощью контропозиции.
3. Доказательство с помощью противоречия (от противного).
4. Доказательство контпримером.
5. Метод математической индукции.

5.1 Использование теоремы о дедукции

Теорема о дедукции служит обоснованием следующего приема, который часто используют в математических доказательствах. Для того, чтобы доказать утверждение «Если A, то B» предполагают, что справедливо A и доказывают справедливость B. Теорема о дедукции справедлива для исчисления высказываний, но оказывается, что она остается в силе и для теории первого порядка. Точнее, если формулы исчисления предикатов первого порядка A и B замкнуты (тогда в любой интерпретации их истинность вполне определена), то для этих формул справедливо утверждение теоремы о дедукции.

Пример. Докажите, что для каждого целого n, если n четное, то n^2 тоже четное.

Доказательство. Так как n четное, то его можно представить в виде $n=2m$, где m – целое число. Поэтому, $n^2 = (2m)^2 = 4\ m^2 = 2\ (2\ m^2)$, где $2\ m^2$ – целое число, т.е. n^2 четное.

5.2 Доказательство импликаций с помощью контропозиции

Рассмотрим условное высказывание вида $A \supset B$, где A – конъюнкция посылок, B – заключение. Иногда удобнее вместо доказательства истинности этой импликации установить логическую истинность некоторого другого высказывания, равносильного исходному. Такие формы доказательства называются *косвенными методами* доказательства.

Один из наиболее применяемый косвенный вид доказательства – это доказательство с помощью контропозиции.

Контропозицией формулы $A \supset B$ называется равносильная формула $\neg B \supset \neg A$. Поэтому, если мы установим истинность контропозиции, то тем самым докажем истинность исходной импликации.

Пример. На основе контропозиции докажите, что если m и n – произвольные положительные целые числа такие, что $m \times n \le 100$, то либо $m \le 10$, либо $n \le 10$.

Доказательство. Контропозицией исходному утверждению служит следующее высказывание: «Если $m>10$ и $n>10$, то $m \times n > 100$», что очевидно.

Преимущества метода доказательства с помощью контропозиции проявляются при автоматизированном способе доказательства, т.е. когда доказательство совершает компьютер с помощью специальных программных систем доказательства теорем.

При построении выводов не всегда целесообразно ждать появления искомого заключения, просто применяя правила вывода. Именно такое часто случается, когда мы делаем допущение A для доказательства импликации $A \supset B$. Мы применяем цепное правило и модус поненс к A и другим посылкам, чтобы в конце получить B. Однако можно пойти по неправильному пути, и тогда будет доказано много предложений, большинство из которых не имеет отношения к нашей цели. Этот метод носит название *прямой волны* и имеет тенденцию порождать лавину промежуточных результатов, если его запрограммировать для компьютера и не ограничить глубину.

Другая возможность – использовать контрапозицию и попытаться, например, доказать $\neg B \supset \neg A$ вместо $A \supset B$. Тогда мы допустим $\neg B$ и попробуем доказать $\neg A$. Это позволяет двигаться как бы назад от конца к началу, применяя правила так, что старое заключение играет роль посылки. Такая организация поиска может лучше показать, какие результаты имеют отношение к делу. Она называется *поиском от цели*.

5.3 Доказательство от противного

Частным случаем косвенных методов доказательства является приведение к противоречию (от противного). Метод доказательства основывается на следующем утверждении.

Если Г, $\neg S \vdash F$, где F – любое противоречие (тождественно ложная формула), то Г $\vdash S$.

В этом методе используются следующие равносильности:

$A \supset B \equiv \neg (A \supset B) \supset (C \& \neg C) \equiv (A \& \neg B) \supset (C \& \neg C)$,

$A \supset B \equiv (A \& \neg B) \supset \neg A$,

$A \supset B \equiv (A \& \neg B) \supset B$.

Используя вторую из приведенных равносильностей для доказательства $A \supset B$ мы допускаем одновременно A и $\neg B$, т.е. предполагаем, что заключение ложно:

$\neg (A \supset B) \equiv \neg (\neg A \vee B) \equiv A \ \& \ \neg B$.

Теперь мы можем двигаться и вперед от A, и назад от $\neg B$. Если B выводимо из A, то, допустив A, мы доказали бы B. Поэтому, допустив $\neg B$, мы получим противоречие. Если же мы выведем $\neg A$ из $\neg B$, то тем самым получим противоречие с A. В общем случае мы можем действовать с обоих концов, выводя некоторое предложение C, двигаясь вперед, и его отрицание $\neg C$, двигаясь назад. В случае удачи это доказывает, что наши посылки *несовместимы* или *противоречивы*. Отсюда мы выводим, что дополнительная посылка $A \& \neg B$ должна быть ложна, а значит противоположное ей утверждение $A \supset B$ истинно. Метод доказательство от противного – один из самых лучших инструментов математика. «Это гораздо более “хитроумный” гамбит, чем любой шахматный гамбит: шахматист может пожертвовать пешку или даже фигуру, но математик жертвует *партию*» (Г. Харди).

Пример 1. Докажем, что диагональ единичного квадрата является иррациональным числом.

Доказательство. Используя теорему Пифагора переформулируем утверждение: «Не существуют два таких целых числа p и q, чтобы выполнялось отношение

$$\sqrt{2} = \frac{p}{q}».$$

В самом деле, тогда мы приходим к равенству $p^2 = 2q^2$. Мы можем считать, что дробь p/q несократима, иначе мы с самого начала сократили бы ее на наибольший общий делитель чисел p и q. С правой стороны имеется 2 в качестве множителя, и потому p^2 есть четное число, и, значит, само p – также четное, так как квадрат нечетного числа есть нечетное число. В таком случае можно положить $p = 2r$. Тогда равенство принимает вид:

$$4r^2 = 2q^2, \text{ или } 2r^2 = q^2.$$

Так как с левой стороны теперь имеется 2 в качестве множителя, значит q^2, а следовательно, и q – четное. Итак, и p и q – четные числа, т.е. делятся на 2, а это противоречит допущению, что дробь p/q несократима. Итак, равенство $p^2 = 2q^2$ невозможно, и $\sqrt{2}$ не может быть рациональным числом.

Пример 2. Доказать, что простых чисел бесконечно много.

Доказательство. Предположим, что существует конечное множество простых чисел и p есть наибольшее из них: 2, 3, 5, 7, 11, …, p. Определим число $N=p!+1$. Число N при делении на любое из чисел 2, 3, 5, 7, 11, …, p дает в остатке 1. Каждое число, которое не является простым, делится, по крайней мере, на одно простое число. Число N не делится ни на одно простое число, следовательно, N само простое число, причем $N>p$. Таким образом, мы пришли к противоречию, которое доказывает, что простых чисел бесконечно много.

5.4 Доказательство контрпримером

Многие математические гипотезы имеют в своей основе форму: «Все объекты со свойством A обладают свойством B». Мы можем записать это в виде формулы

$\forall x\ (A(x) \supset B(x))$,

где $A(x)$ обозначает предикат «x обладает свойством A», $B(x)$ – «x обладает свойством B». Если число возможных значений x является конечным, то в принципе доказательство может быть проведено с помощью разбора случаев, то есть непосредственной проверкой выполнимости гипотезы для каждого объекта. В случае если число объектов не является конечным, то такой возможности не существует даже в принципе. Однако для доказательства ложности гипотезы достаточно привести хотя бы один пример (называемый в этом случае *контрпример*), для которого гипотеза не выполнима.

5.5 Математическая индукция

В математических доказательствах часто используется *принцип математической индукции*, который формулируется следующим образом: если какое-то высказывание $P(n)$, зависящее от натурального параметра n, доказано для $n = 0$ и при произвольном n удается из истинности $P(n)$ обосновать истинность $P(n+1)$, то $P(n)$ истинно для всех n.

Приведем пример доказательства по индукции. *Доказать, что сумма трех последовательных кубов натуральных чисел делится на 9*.

Базис индукции. $1^3+2^3+3^3 = 36$ и делится на 9.

Индуктивный переход. Что произвести индуктивный переход, нужно предположить доказываемое утверждение для n и затем доказать его для $n+1$. Пусть $n^3 + (n+1)^3 + (n+2)^3$ делится на 9. Тогда

$$(n+1)^3 + (n+2)^3 + (n+3)^3 = (n+1)^3 + (n+2)^3 + n^3 + 9n^2 + 27n + 27 =$$
$$(n^3+(n+1)^3 + (n+2)^3)+(9n^2 + 27n + 27).$$

Но все слагаемые в последней скобке делятся на 9, а первая скобка делится на 9 по предположению, значит, и исходная сумма делится на 9. Таким образом, мы установили, что делимость суммы, начинающейся с n, влечет делимость суммы, начинающейся с $n+1$.

Следовательно, утверждение доказано для всех n.

Итак, в математической индукции имеется *индуктивный базис* – утверждение, что свойство выполнено для самого маленького из рассматриваемых чисел, и *индуктивный переход* – обоснование перехода от числа n к числу $n+1$.

Математическая индукция
$A(0)\ \&\ \forall n\ (A(n) \supset A(n+1)) \supset \forall n\ A(n)$

Принцип математической индукции допускает несколько переформулировок, которые в классической логике эквивалентны исходному принципу.

Первая из них – *возвратная индукция*. Здесь переход происходит не от одного значения к следующему, а от всех предыдущих значений к последующему, шаг индукции переводит не от $A(n)$ к $A(n+1)$, а от $\forall x<n\ A(x)$ к $A(n)$. Как ни парадоксально, при таком переходе не требуется базиса индукции. В самом деле, поскольку условие $x<0$ тождественно ложно, то, поскольку из лжи следует все что угодно, имеем $\forall x(x<0 \supset A(x))$, а отсюда по индуктивному переходу имеем $A(0)$.

Возвратная индукция
$\forall x(\forall y(y<x \supset A(y)) \supset A(x)) \supset \forall x\ A(x)$

Еще одна переформулировка метода математической индукции была известна еще древним грекам. Это – *метод бесконечного спуска*.

Если для каждого натурального числа, удовлетворяющего свойству $A(n)$, найдется меньшее, удовлетворяющее этому свойству, то чисел n, для которых выполнено $A(n)$, вообще нет.

Бесконечный спуск

$$\forall n(A(n) \supset \exists m(m<n \ \& \ A(m))) \supset \forall n \neg A(n)$$

Рассмотрим пример применения метода бесконечного спуска. Докажем, что абсурдно предположение, что у каждого человека мать является человеком. В самом деле, за все время существования Земли на ней жило конечное число людей. Пусть у каждого человека мать – человек. Упорядочим людей по моментам рождения. Список всех бывших людей в порядке времени рождения назовем Книгой Судеб. Мать рождается раньше своих детей, и поэтому в Книге Судеб она стоит раньше. Таким образом, для каждого человека найдется стоящий раньше него в Книге Судеб. По принципу бесконечного спуска получаем, что людей вообще нет и не было, что абсурдно. Полученное противоречие доказывает утверждение.

Применение математической индукции, конечно же, содержит много тонкостей. Приведем софизм, доказываемый при помощи неправильного применения индукции.

Задача. **Все лошади одной масти**. То, что все лошади одной масти, можно доказать индукцией по числу лошадей в определенном табуне.

Если существует только одна лошадь, то она своей масти, так что база индукции тривиальна. Для индуктивного перехода предположим, что существует n лошадей (с номерами от 1 до n). По индуктивному предположению лошади с номерами от 1 до n–1 одинаковой масти и, аналогично, лошади с номерами от 2 до n имеют одинаковую масть. Но лошади посередине с номерами с 2 до n–1 не могут изменять масть в зависимости от того, как они сгруппированы – это лошади, а не хамелеоны. Поэтому лошади с номерами от 1 до n также должны быть одинаковой масти. Таким образом, все n лошадей одинаковой масти. Что и требовалось доказать.

С методом математической индукции связано понятие индуктивного определения.

Базис индукции. Выражение вида A есть B.

Индуктивный переход. Если мы имеем выражения A_1, …, A_n типа B, то C, построенное из них, также есть выражение типа B.

С каждым индуктивным определением связан *принцип индукции по построению объекта типа B*.

Базис индукции. Каждый объект вида A обладает свойством θ.

Индуктивный переход. Если A_1, …, A_n обладают свойством θ, то и C им обладает.

Заключение индукции. Тогда любой объект типа B обладает свойством θ.

Этот принцип является логическим выражением следующего неявного пункта, присутствующего в любом индуктивном определении: никаких других объектов типа B, кроме полученных применением правил его определения, нет. Иными словами, множество объектов типа B – минимальное из тех, которые включают базисные объекты и замкнуты относительно индуктивного перехода. В простых определениях эту минимальность можно выразить следующим образом: объект должен получаться из базисных конечным числом применений шагов определения.

Стоит отметить, что логическая индукция по построению вовсе не требует однозначности представления объекта в форме, соответствующей одному из пунктов его определения. Но для задания функций индукцией по построению такая однозначность необходима.

В качестве примера, приведем индуктивное определение множества термов и формул языка первого порядка.

Определение термов:

а) переменные и константы суть (атомарные) термы;

б) если f – операция местности n, а $t_1, t_2, ..., t_n$ – термы, то $f(t_1, t_2, ..., t_n)$ – терм.

Определение формул:

а) если P – предикат местности n, а $t_1, t_2, ..., t_n$ – термы, то $P(t_1, t_2, ..., t_n)$ есть (атомарная) формула;

б) если P, Q суть формулы, x – переменная, то выражения $(P)\sim(Q)$, $(P)\supset(Q)$, $(P)\vee(Q)$, $(P)\wedge(Q)$, $\neg(P)$, $\forall x(P)$, $\exists x(P)$ суть формулы.

6 Автоматическое доказательство теорем

6.1 Постановка задачи

Алгоритм, который проверяет отношение

$$\Gamma \vdash_{\boldsymbol{T}} S$$

для формулы S, множества формул Γ, и теории $\boldsymbol{T}$ называется алгоритмом *автоматического доказательства теорем*. В общем случае такой алгоритм невозможен, т. е. не существует алгоритма, который для любых S, Γ и $\boldsymbol{T}$ выдавал бы ответ «Да», если $\Gamma \vdash_{\boldsymbol{T}} S$, и ответ «Нет», если неверно $\Gamma \vdash_{\boldsymbol{T}} S$. Более того, известно, что нельзя построить алгоритм автоматического доказательства теорем даже для большинства конкретных достаточно сложных формальных теорий $\boldsymbol{T}$. В некоторых случаях удается построить алгоритм автоматического доказательства теорем, который применим не ко всем формулам теории (т. е. частичный алгоритм).

Для некоторых простых формальных теорий (например, исчисления высказываний) и некоторых простых классов формул (например, теорий первого порядка с одним одноместным предикатом) алгоритмы автоматического доказательства теорем известны.

Пример. Поскольку для исчисления высказываний известно, что теоремами являются тавтологии, можно воспользоваться простым методом проверки общезначимости формулы с помощью таблиц истинности. А именно, достаточно вычислить истинностное значение формулы при всех возможных интерпретациях (их конечное число). Если во всех случаях получится значение И, то проверяемая формула – тавтология, и, следовательно, является теоремой исчисления высказывания. Если же хотя бы в одном случае получится значение Л, то проверяемая формула не является тавтологией и, следовательно, не является теоремой теории высказываний.

Приведенный выше пример является алгоритмом автоматического доказательства теорем в теории исчисления высказываний, хотя и не является алгоритмом автоматического поиска вывода теорем из аксиом теории исчисления высказываний.

Первые компьютерные реализации систем автоматического доказательства теорем появились в конце 50-х годов, а в 1965г. Робинсон предложил свой *метод резолюций*, который и по сей день лежит в основе большинства систем поиска логического вывода.

Робинсон пришел к заключению, что правила вывода, которые следует применять при автоматизации процесса доказательства с помощью компьютера, не обязательно должны совпадать с правилами вывода, используемыми человеком. Он обнаружил, что общепринятые правила вывода, например, правило *modus ponens*, специально сделаны «слабыми», чтобы человек мог интуитивно проследить за каждым шагом процедуры доказательства. Правило резолюции более сильное, оно трудно поддается восприятию человеком, но эффективно реализуется на компьютере.

Для любой теории первого порядка $\boldsymbol{T}$, любой формулы S и множества формул Γ теории $\boldsymbol{T}$ метод резолюций выдает ответ «Да», если $\Gamma \vdash_{\boldsymbol{T}} S$, и выдает ответ «Нет» или не выдает никакого ответа (т.е. зацикливается), если неверно $\Gamma \vdash_{\boldsymbol{T}} S$.

В основе метода резолюций лежит идея «доказательства от противного»: пытаемся доказать $\Gamma, \neg S \vdash F$, где F – любое противоречие. Отсюда будет следовать $\Gamma \vdash_{\boldsymbol{T}} S$.

Пустая формула не имеет никакого значения ни в какой интерпретации, в частности, не является истинной ни в какой интерпретации и, по определению, является противоречием. В качестве формулы F при доказательстве от противного по методу резолюций принято использовать пустую формулу, которая обозначается $\square$.

6.2 Предваренная нормальная форма

Приведение формулы к виду, пригодному для применения метода резолюций, требует рассмотрение некоторых нормальных форм формул.

Формула $Q_1x_1 \ldots Q_nx_n\, A$, где символ Q_i означает либо $\forall$, либо $\exists$, x_i и x_j различны для $i \neq j$ и A – не содержит кванторов, называется формулой в *предваренной нормальной форме.* (Сюда включается и случай n=0, когда вообще нет никаких кванторов.)

Без ограничения общности можно потребовать, чтобы квантифицированы могли быть только переменные, имеющие свободное вхождение. Действительно, по правилам интерпретации если формула A не содержит вхождений переменной x, то формулы A, $\exists xA$ и $\forall xA$ имеют одно и то же значение истинности при всех интерпретациях.

Интерес к предваренным формам связан со следующей теоремой, справедливой для всякой теории первого порядка,

Теорема 7. Существует алгоритм, преобразующий всякую формулу A к такой формуле B в предваренной нормальной форме, что $A=B$.

Доказательство. Алгоритм получения предваренной формулы очень прост. Этапы таковы:

- Исключить связки эквивалентности и импликации по правилам:
  $A \sim B \equiv (A \supset B)\&(B \supset A) \equiv (A\&B) \vee (\neg A\&\neg B)$;
  $A \supset B \equiv \neg A \vee B \equiv \neg\, (A\&\neg B)$.
- Переименовать (если необходимо) связанные переменные таким образом, чтобы никакая переменная не имела бы одновременно свободных и связанных вхождений. Это условие требуется не только для рассматриваемой формулы, но также для всех ее подформул.
- *Разделение связанных переменных*. Логически эквивалентное преобразование: $Q_1x\, A(\ldots Q_2x\, B(\ldots x\ldots)\ldots) = Q_1x\, A(\ldots Q_2y\, B(\ldots y\ldots)\ldots)$, где Q_1 и Q_2 – любые кванторы и y – любая переменная, не входящая в формулу $B(\ldots x\ldots)$.
- Удалить те квантификации, область действия которых не содержит вхождений квантифицированной переменной. Такие квантификации в действительности не нужны.
- *Протаскивание отрицаний*. Преобразовать все вхождения отрицания в стоящие непосредственно перед элементарными подформулами в соответствии со следующими правилами:
  $\neg\forall xA = \exists x\neg A$;
  $\neg\exists xA = \forall x\neg A$;
  $\neg\neg A = A$;
  $\neg(A \vee B) = \neg A\&\neg B$;
  $\neg(A\&B) = \neg A \vee \neg B$.
- *Перемещение кванторов.* С помощью логически эквивалентных преобразований перемещаем все квантификации в начало формулы. Для этого используется следующий набор правил преобразования, которые справедливы во всякой теории первого порядка. Здесь мы приведем эти правила для конъюнкции. Парные правила для дизъюнкции выглядят аналогично.
  $(\forall x\, A\ \&\ \forall x\, B) = \forall x\, (A\&B)$.
  $(\forall x\, A\ \&\ B) = \forall x\, (A\&B)$, если B не содержит x.
  $(A\ \&\ \forall x\, B) = \forall x\, (A\&B)$, если A не содержит x.
  $(\exists x\, A\ \&\ B) = \exists x\, (A\&B)$, если B не содержит x.
  $(A\ \&\ \exists x\, B) = \exists x\, (A\&B)$, если A не содержит x.

  Эти правила позволяют в каждой формуле постепенно передвигать все кванторы влево.

  Для полноты этого набора правил необходимо добавить возможность переименования некоторых связных переменных. Например, формула $\exists xA(x)\ \&\ \forall xB(x)$ будет сначала преобразована в $\exists xA(x)\ \&\ \forall yB(y)$ (перед применением правил преобразования). Заметим также, что для упрощения формул в

любой момент можно применять свойства коммутативности, ассоциативности и идемпотентности связок & и ∨. Итак, теорема конструктивно доказана.

Пример. Найдем предваренную форму, эквивалентную формуле

$$\forall x(P(x) \,\&\, \forall y \exists x(\neg Q(x,y) \supset \forall z R(a,x,y))).$$

Этапы этого поиска последовательно перечислены ниже.

Исключение связки импликации:

$$\forall x(P(x) \,\&\, \forall y \exists x(\neg\neg Q(x,y) \vee \forall z R(a,x,y))).$$

Разделение связных переменных:

$$\forall x(P(x) \,\&\, \forall y \exists u(\neg\neg Q(u,y) \vee \forall z R(a,u,y))).$$

Удаление бесполезной квантификации:

$$\forall x(P(x) \,\&\, \forall y \exists u(\neg\neg Q(u,y) \vee R(a,u,y))).$$

Протаскивание отрицаний:

$$\forall x(P(x) \,\&\, \forall y \exists u(Q(u,y) \vee R(a,u,y))).$$

Перемещение кванторов:

$$\forall x \forall y\ (P(x) \,\&\, \exists u(Q(u,y) \vee R(a,u,y))),$$
$$\forall x \forall y\ \exists u\ (P(x) \,\&\, (Q(u,y) \vee R(a,u,y))).$$

Замечание. Одна формула может допускать много эквивалентных предваренных форм. Вид полученного результата зависит от порядка применения правил, а также от произвола при переименовании.

6.3 Сколемизация

Рассмотрим один из математических результатов, послуживших идейной основой метода резолюций. Интуитивно сочетание кванторов $\forall x \exists y$ может пониматься как утверждение о существовании функции, строящей y по x. Рассмотрим несколько примеров. Если замкнутая формула $\exists y P(y)$ выполнима в некоторой интерпретации, то существует некоторая предметная константа c из универсума, для которой формула $P(c)$ истинна. Другой случай. Пусть, например, имеется формула P с двумя параметрами x и y. Тогда замкнутая формула $\forall x \exists y P(x,y)$ выполнима тогда и только тогда, когда выполнима формула $\forall x P(x, f(y))$, где f – новый одноместный функциональный символ.

Аналогичное преобразование выполнимо и для произвольных предваренных формул. Например, формула

$$\forall x \forall y \exists z \forall u \exists v\ P(x,y,z,u,v)$$

выполнима тогда и только тогда, когда выполнима формула

$$\forall x \forall y \forall u\ P(x,y, f(x,y),u,g(x,y,u))$$

(здесь f и g – функциональные символы, не встречающиеся в формуле P).

Элиминация кванторов существования (сколемизация)

Сколемовской формой называется предваренная нормальная форма, не содержащая кванторы существования.

Сколемизация осуществляется преобразованиями:

$\exists x_1\ Q_2 x_2 \ldots\ Q_n x_n\ A(x_1, x_2, \ldots, x_n) \to Q_2 x_2 \ldots\ Q_n x_n\ A(a, x_2, \ldots, x_n)$;

$\forall x_1 \ldots\ \forall x_{i-1}\ \exists x_i\ Q_{i+1} x_{i+1} \ldots\ Q_n x_n\ A(x_1, \ldots, x_i, \ldots, x_n) \to$
$\forall x_1 \ldots\ \forall x_{i-1}\ Q_{i+1} x_{i+1} \ldots\ Q_n x_n\ A(x_1, \ldots, f(x_1, \ldots, x_{i-1}), \ldots, x_n)$,

где a – новая предметная константа, f – новый функтор, а $Q_2, \ldots, Q_n$ – любые кванторы, символ $\to$ показывает результат преобразования.

Функции для замены кванторов существования впервые стал использовать Т. Сколем.

Теорема 8. Пусть даны замкнутая формула A и формула B – результат сколемизации A. Тогда формулы A и B одновременно выполнимы или невыполнимы.

6.4 Конъюнктивная нормальная форма

Литерал – это элементарная формула или отрицание элементарной формулы.
Дизъюнкт – это литерал или дизъюнкция конечного числа литералов.
Конъюнктивная нормальная форма – это формула, которая являются дизъюнктом или конъюнкцией конечного числа дизъюнктов.

Теорема 9. Для любой формулы A, не содержащей кванторы, существует формула B, являющаяся конъюнктивной нормальной формой (КНФ), и B логически эквивалентна A.

Доказательство. Укажем алгоритм построения формулы B.

- Исключаем связки эквивалентности и импликации по правилам:
  $X \sim Y = (X \supset Y)\&(Y \supset X)$;
  $X \supset Y = \neg X \vee Y$.
- *Протаскивание отрицаний*. Преобразуем все вхождения отрицания в стоящие непосредственно перед элементарными подформулами в соответствии со следующими правилами:
  $\neg\neg X = X$;
  $\neg(X \vee Y) = \neg X \& \neg Y$;
  $\neg(X \& Y) = \neg X \vee \neg Y$.
- Преобразуем полученную формулу, используя правило дистрибутивности $\vee$ относительно $\&$.

Пример. Приведем к КНФ формулу $(X\&Y)\sim(\neg X\&Z)$.

- После первого этапа получаем

$(X\&Y)\sim(\neg X\&Z) = (X\&Y \supset \neg X\&Z) \ \& \ (\neg X\&Z \supset X\&Y) =$
$(\neg(X\&Y) \vee (\neg X\&Z)) \ \& \ (\neg(\neg X\&Z) \vee (X\&Y))$.

- После второго этапа получаем

$(\neg(X\&Y) \vee (\neg X\&Z)) \ \& \ (\neg(\neg X\&Z) \vee (X\&Y)) = ((\neg X \vee \neg Y) \vee (\neg X\&Z)) \ \& \ ((\neg\neg X \vee \neg Z) \vee (X\&Y))$
$= ((\neg X \vee \neg Y) \vee (\neg X\&Z)) \ \& \ ((X \vee \neg Z) \vee (X\&Y))$.

- После третьего этапа получаем

$((\neg X \vee \neg Y) \vee (\neg X\&Z)) \ \& \ ((X \vee \neg Z) \vee (X\&Y)) =$
$(((\neg X \vee \neg Y) \vee \neg X) \ \& \ ((\neg X \vee \neg Y) \vee Z)) \ \& \ ((X \vee \neg Z) \vee (X\&Y)) =$
$(((\neg X \vee \neg Y) \vee \neg X) \ \& \ ((\neg X \vee \neg Y) \vee Z)) \ \& \ (((X \vee \neg Z) \vee X) \ \& \ ((X \vee \neg Z) \vee Y))$.

- Используя идемпотентность и ассоциативность связок, упрощаем результат:

$(\neg X \vee \neg Y) \ \& \ (\neg X \vee \neg Y \vee Z) \ \& \ (X \vee \neg Z)\&(X \vee \neg Z \vee Y)$.

Заметим, что в силу ассоциативности операций $\&$ и $\vee$, как бы мы не расставляли скобки в выражениях $A_1\&A_2\&\ldots\&A_n$ и $A_1 \vee A_2 \vee \ldots \vee A_n$ (n>2), всегда будем приходить к логически эквивалентным формулам. Кроме того, КНФ не являются однозначно определенным. Формула может иметь несколько равносильных друг другу КНФ.

6.5 Сведение к дизъюнктам

Метод резолюций работает с дизъюнктами. Любая формула A исчисления предикатов может быть преобразована в множество дизъюнктов следующим образом (здесь знак $\rightarrow$ используется для обозначения способа преобразования формул).

1. *Приводим к предваренной форме.* См. алгоритм в доказательстве теоремы 7. Получаем формулу A_1 логически эквивалентную формуле A.
2. *Проводим сколемизацию* формулы A_1 и получаем формулу A_2. По теореме 8 формулы A_1 и A_2 одновременно выполнимы или невыполнимы.
3. *Элиминация кванторов всеобщности.* Выполняем преобразования вида: $\forall x \ P(x) \rightarrow P(x)$. Это преобразование не является логически эквивалентным, но формула $\forall x P(x) \supset P(x)$ является логически общезначимой. После третьего этапа получаем фор-

мулу A_3, которая не содержит кванторов и формулы A_2 и A_3 одновременно выполнимы или невыполнимы.

4. *Приведение к конъюнктивной нормальной форме*. Выполняем логически эквивалентные преобразования вида: $X \vee (Y \& Z) = (X \vee Y) \& (X \vee Z)$; После четвертого этапа получаем формулу A_4, которая находится в конъюнктивной нормальной форме.
5. *Элиминация конъюнкции*. Преобразование: $X \& Y \to X, Y$. После пятого этапа формула распадается на множество дизъюнктов.

Теорема 10. Если Г – множество дизъюнктов, полученных из формулы A, то A является противоречием тогда и только тогда, когда множество Г невыполнимо.
(Множество формул Г невыполнимо – это означает, что множество Г не имеет модели, то есть не существует интерпретации, в которой бы все формулы Г имели бы значение И.)

Доказательство. Формулы A и A_4 одновременно выполнимы или невыполнимы в любой интерпретации. Формула A_4 является противоречием тогда и только тогда, когда Г не имеет модели.

6.6 Правило резолюции для исчисления высказываний

Пусть C_1 и C_2 – два дизъюнкта в исчислении высказываний, и пусть $C_1 = P \vee A$, а $C_2 = \neg P \vee B$, где P – пропозициональная переменная, а A, B – любые дизъюнкты (в частности, может быть, пустые или состоящие только из одного литерала). Правило вывода

$$\frac{C_1, C_2}{A \vee B} R$$

называется *правилом резолюции.* Дизъюнкты C_1 и C_2 называются *резольвируемыми* (или *родительскими*), дизъюнкт $A \vee B$ – *резольвентой*, а формулы P и $\neg P$ – *контрарными* литералами.

Правило резолюции – это очень мощное правило вывода. Многие правила вывода являются частными случаями правила резолюции:

$$\frac{A, A \supset B}{B} \textit{Modus ponens} \qquad \frac{A, \neg A \vee B}{B} R$$

$$\frac{A \supset B, B \supset C}{A \supset C} \textit{Транзитивность} \qquad \frac{\neg A \vee B, \neg B \vee C}{\neg A \vee C} R$$

Доказательство следующей теоремы тривиально.

Теорема 11. Резольвента является логическим следствием резольвируемых дизъюнктов.

6.7 Унификация

Пусть имеются две элементарные формулы A и B языка первого порядка. Эти формулы могут содержать индивидные (предметные) переменные. В некоторых случаях возможна такая подстановка термов вместо переменных в этих формулах, что формулы A и B (вообще говоря, различные) становятся тождественными. Такая подстановка называется *унификатором*. Например, формулы $F(x, p(x,y))$ и $F(t(z,c),u)$ можно унифицировать: для этого достаточно переменную x заменить на $t(z,c)$, а вместо переменной u подставить терм $p(t(z,c),y)$; в результате унификации получаем формулу $F(t(z,c), p(t(z,c),y))$.

В общем случае под *подстановкой* мы понимаем множество $\theta = \{t_1/X_1, ..., t_n/X_n\}$, где все X_i (i=1,..., n) – различные переменные, а t_i – термы, отличные от всех X. Результат применения подстановки θ к элементарной формуле E (т. е. результат одновременной замены всех переменных X_1, ..., X_n на термы t_1, ..., t_n соответственно) обозначается $E\theta$. Композиция $\theta\lambda$ подстановок θ и λ определяется естественным образом и удовлетворяет условию $(E\theta)\lambda = E(\theta\lambda)$ для любой формулы E.

Унификатором формул E_1 и E_2 называется такая подстановка θ, что $E_1\theta=E_2\theta$. Вообще говоря, у двух формул E_1 и E_2 может быть бесконечно много унификаторов, однако среди них всегда имеется *наиболее общий* (или самый общий) унификатор σ, такой, что все остальные унификаторы получаются в результате действия на σ некоторыми подстановками.

6.8 Правило резолюции для исчисления предикатов

Для применения правила резолюции нужны контрарные литералы в резольвируемых дизъюнктах. Пусть C_1 и C_2 – два дизъюнкта в исчислении предикатов. Правило вывода

$$\frac{C_1, C_2}{(A \vee B)\sigma} R$$

называется правилом резолюции в исчислении предикатов, если в дизъюнктах C_1 и C_2 существуют унифицируемые контрарные литералы P_1 и P_2, то есть $C_1 = P_1 \vee A$, а $C_2 = \neg P_2 \vee B$, причем атомарные формулы P_1 и P_2 являются унифицируемыми наиболее общим унификатором σ. В этом случае резольвентой дизъюнктов C_1 и C_2 является дизъюнкт $(A \vee B)\sigma$, полученный из дизъюнкта $A \vee B$ применением унификатора σ.

6.9 Алгоритм резолюций

Мы можем попробовать доказать невыполнимость конечного множества дизъюнктов Γ с помощью следующего алгоритма резолюций ($\square$ обозначает пустой дизъюнкт).

while $\square \notin \Gamma$ do

begin

- выбираем дизъюнкты C и D из Γ, содержащие унифицируемые контрарные литералы;
- вычисляем резольвенту R;
- заменяем Γ на $\Gamma \cup \{R\}$.

end

Теорема 12. Если множество дизъюнктов является невыполнимым, то алгоритм резолюций за конечное число шагов придет к противоречию (пустому дизъюнкту).

В качестве примера проверим невыполнимость множества дизъюнктов

$$\Gamma = \{P \vee Q, P \vee R, \neg Q \vee \neg R, \neg P\}.$$

Дизъюнкты удобно пронумеровать. Получится следующий список:

1. $P \vee Q$
2. $P \vee R$
3. $\neg Q \vee \neg R$
4. $\neg P$

Затем вычисляются и добавляются к Γ резольвенты. В списке, который приводится ниже, каждый дизъюнкт – резольвента некоторых из предшествующих дизъюнктов. Номера их приведены в скобках справа от соответствующей резольвенты.

5. $P \vee \neg R$ (1,3)
6. Q (1,4)
7. $P \vee \neg Q$ (2,3)
8. R (2,4)
9. P (2,5)
10. $\neg R$ (3,6)
11. $\neg Q$ (3,8)
12. $\neg R$ (4,5)
13. $\neg Q$ (4,7)
14. $\square$ (4,9)

При работе алгоритма возможны три случая.

1. Среди текущего множества дизъюнктов нет резольвируемых. Это означает, что множество формул Г выполнимо.

2. В результате очередного применения правила резолюции получен пустой дизъюнкт. Это означает, что множество формул Г невыполнимо

3. Процесс не заканчивается, то есть множество дизъюнктов пополняется все новыми резольвентами, среди которых нет пустых. Выполнение алгоритма не завершится, например, для множества $\{P, \neg P \vee Q\}$. Резольвентный дизъюнкт Q будет порождаться неограниченное число раз.

Алгоритм проверки невыполнимости – недетерминированный: вообще говоря, возможен не один выбор для резольвируемых дизъюнктов и контрарных литералов. В приведенном примере мы выбирали дизъюнкты C и D в лексикографическом порядке номеров. Такая стратегия неоптимальная. Некоторые резольвенты оказались ненужными, а также вычислялись в некоторых случаях более одного раза. Для сравнения продемонстрируем теперь применение этого же алгоритма с минимумом резолюций:

5. Q (1, 4)
6. R (2, 4)
7. $\neg Q$ (3, 6)
8. □ (5, 7)

Конкретные реализации выбора резольвируемых дизъюнктов и контрарных литералов называются *стратегиями* метода резолюции.

6.10 Опровержение методом резолюций

Опровержение методом резолюций – это алгоритм автоматического доказательства теорем в прикладном исчислении предикатов, который сводится к следующему. Пусть нужно установить выводимость $\Gamma \vdash G$.

Каждая формула множества Г и формула $\neg G$ (отрицание целевой теоремы) независимо преобразуется во множества дизъюнктов. Пусть C – полученное совокупное множество дизъюнктов.

Теорема 13. Множество дизъюнктов C невыполнимо тогда и только тогда, когда $\Gamma \vdash G$.

С помощью алгоритма резолюций, примененному к множеству дизъюнктов C, мы можем установить выводимость $\Gamma \vdash G$.

При работе алгоритма возможны три случая.

1. На каком-то этапе отсутствуют резольвируемые дизъюнкты. Это означает, что теорема опровергнута, то есть формула G не выводима из множества формул Г.

2. В результате очередного применения правила резолюции получен пустой дизъюнкт. Это означает, что теорема доказано, то есть $\Gamma \vdash G$.

3. Процесс не заканчивается, то есть множество дизъюнктов пополняется все новыми резольвентами, среди которых нет пустых. Это ничего не означает.

Замечание. Таким образом, исчисление предикатов является полуразрешимой теорией, а метод резолюций является частичным алгоритмом автоматического доказательства теорем.

Пример. Докажем методом резолюций теорему $\vdash (((A \supset B) \supset A) \supset A)$. Сначала нужно преобразовать в дизъюнкты отрицание целевой формулы $\neg(((A \supset B) \supset A) \supset A)$. Используя правила из пункта «Сведения к дизъюнктам», получаем множество дизъюнктов $\{A \vee A, \neg B \vee A, \neg A\}$.

После этого проводится резольвирование имеющихся предложений 1–3.

1. $A \vee A$.
2. $\neg B \vee A$.
3. $\neg A$.
4. A (1, 3)
5. □ (3, 4).

Таким образом, теорема доказана.

В настоящее время предложено множество различных стратегий метода резолюций. Среди них различаются *полные* и *неполные* стратегии. Полные стратегии – это такие, которые гарантируют нахождение доказательства теоремы, если оно вообще существует. Неполные стратегии могут в некоторых случаях не находить доказательства, зато они работают быстрее. Следует иметь в виду, что автоматическое доказательство теорем методом резолюций имеет по существу переборный характер, и этот перебор столь велик, что может быть практически неосуществим за приемлемое время.

При автоматическом доказательстве теорем методом резолюций основная доля вычислений приходится на поиск унифицируемых дизъюнктов. Таким образом, эффективная реализация алгоритма унификации критически важна для практической применимости метода резолюций.

Рассмотрим следующий пример из [2], показывающий как можно получить ответы на вопросы с помощью метода резолюций. Предположим, что у нас есть предикат $F(x,y)$, означающий, что «x – отец y», и истинна следующая формула

$F(john,harry)$ & $F(john,sid)$ & $F(sid,liz)$.

Таким образом, у нас есть три дизъюнкта. Они не содержат переменных или импликаций, а просто представляют базисные факты.

Введем еще три предиката $M(x)$, $S(x,y)$ и $B(x,y)$, означающие соответственно, что x – мужчина, что он единокровен с y, что он брат y. Мы можем записать следующие аксиомы о семейных отношениях:

$\forall x,y\ (F(x,y) \supset M(x))$;

$\forall x,y,w\ ((F(x,y)\ \&\ F(x,w)) \supset S(y,w))$;

$\forall x,y\ ((S(x,y)\ \&\ M(x)) \supset B(x,y))$.

Они утверждают следующее: 1) все отцы – мужчины; 2) если у детей один отец, то они единокровны; 3) брат – это единокровный мужчина.

Пусть мы задали вопрос $\exists z\ B(z,harry)$? Чтобы найти ответ с помощью метода резолюции, мы записываем отрицание вопроса $\forall z \neg B(z,harry)$. Затем приводим аксиомы к нормальной форме и записываем каждый дизъюнкт в отдельной строке (так как каждый дизъюнкт истинен сам по себе):

1. $\neg F(x,y) \vee M(x)$;
2. $\neg F(x,y) \vee \neg F(x,w) \vee S(y,w)$;
3. $\neg S(x,y) \vee \neg M(x) \vee B(x,y)$;
4. $F(john,harry)$;
5. $F(john,sid)$;
6. $F(sid,liz)$;
7. $\neg B(z,harry)$.

Для применения резолюции необходимо найти для данной пары дизъюнктов такую подстановку термов вместо переменных, чтобы после нее некоторый литерал одного из дизъюнктов стал отличаться от некоторого литерала другого дизъюнкта лишь отрицанием. Если, например, мы подставим *john* вместо x и *sid* вместо y, то получим следующее:
$\neg F(john,sid) \vee \neg F(john,w) \vee S(sid,w)$.

Мы можем применить правило резолюции к этому дизъюнкту и к (5), что дает новый дизъюнкт

8. $\neg F(john,w) \vee S(sid,w)$ (5, 2) $\{x \to john, y \to sid\}$

Продолжая, получим

9. $S(sid,harry)$ (4, 8) $\{w \to harry\}$
10. $M(sid)$ (6, 1) $\{x \to sid, y \to liz\}$
11. $\neg S(sid,y) \vee B(sid,y)$ (10, 3) $\{x \to sid\}$
12. $B(sid,harry)$ (9, 11) $\{y \to harry\}$
13. □ (12, 7) $\{z \to sid\}$

Таким образом, мы вывели дизъюнкт (12), выражающий, что *sid* брат *harry*, используя аксиомы и факты (4), (5) и (6). Это противоречит отрицанию нашего вопроса, которое утверждает, что *harry* не имеет братьев.

7 Логическое программирование

Методология логического программирования – подход, согласно которому программа содержит описание проблемы в терминах фактов и логических формул, а решение проблемы система выполняет с помощью механизма логического вывода.

Идея использования языка логики предикатов первого порядка в качестве языка программирования возникла еще в 60-ые годы, когда создавались многочисленные системы автоматического доказательства теорем и основанные на них вопросно-ответные системы. Суть этой идеи заключается в том, чтобы программист не указывал машине последовательность шагов, ведущих к решению задачи, как это делается во всех процедурных языках программирования, а описывал на логическом языке свойства интересующей его области, иначе говоря, описывал мир своей задачи. Другие свойства и удовлетворяющие им объекты машина находила бы сама путем построения логического вывода.

После открытия метода резолюций он, по предложению Грина (Cordell Green), вскоре был использован в качестве основы нового языка программирования. Алан Колмероэ (Alain Colmerauer) создал язык логического программирования Prolog в 1971 году. В основе логических языков лежит теория хорновских дизъюнктов. Логическое программирование пережило пик популярности в середине 80-х годов 20 века, когда оно было положено в основу проекта разработки программного и аппаратного обеспечения вычислительных систем пятого поколения [11].

Можно выделить два основных метода, присущих методологии логического программирования:

- метод единообразного применения механизма логического доказательства ко всей программе;
- метод унификации – механизм сопоставления с образцом для создания и декомпозиции структур данных.

7.1 Стратегия метода резолюций в Прологе

Поиск ответов на запросы к программам на Прологе осуществляется путем построения логического вывода в системе *SLD*-резолюции. Прежде чем дать формальное определение *SLD*–резолюции введем некоторые понятия [4, с. 359–361].

Как уже говорили, все переменные, содержащиеся в фактах и правилах, находятся в области действия (явно не указанных) кванторов всеобщности. Поэтому, не меняя смысла программы, мы можем изменять имена переменных, избегая тем самым их коллизии при унификации. *Переименованием* переменных в выражении E_1 относительно выражения E_2 называется любая подстановка вида $\theta = \{Y_1/X_1,\ldots, Y_n/X_n\}$ такая, что $Y_1,\ldots, Y_n$ – различные переменные не входящие в E_1, и выражения $E_1\theta$ и E_2 общих переменных не имеют.

Состоянием будем называть всякую пару вида $<G, \theta>$, где G - запрос, а θ - подстановка (пустой запрос принято обозначать через □, а пустую подстановку - через ε). Для каждой программы ***P*** определим на множестве состояний бинарное отношение $\rightarrow$. Допустим, что G имеет вид $A_1, .., A_n$ и в программе ***P*** присутствует правило

B:-B_1, ..., B_k.

(или факт B, если k=0). Пусть далее θ_1 - переименование $B, B_1, ..., B_k$ относительно G, и пусть $\theta_2 = \{t_1/X_1, ..., t_n/X_n\}$ - наиболее общий унификатор A_i и $B\theta_1$ при некотором i ($1 \le i \le n$), причем термы t_i и формулы A_i не имеют общих переменных. В этом случае мы пишем

$<(A_1, .., A_n), \theta> \rightarrow <(A_1, ..., A_{i-1}, B_1\theta_1, ..., B_k\theta_1, A_{i+1}, ..., A_n)\ \theta_2,\ \theta\theta_2>$

и называем запрос

$(A_1, ..., A_{i-1}, B_1\theta_1, ..., B_k\theta_1, A_{i+1}, ..., A_n)\ \theta_2$

резольвентой запроса G и указанного выше правила.

Будем говорить, что состояние S_1 *переводится* программой ***P*** в состояние S_k (обозначение $S_1 \rightarrow^* S_k$), если существует такая последовательность состояний S_2, ..., S_{k-1}, что $S_i \rightarrow S_{i+1}$ для любого $i \le k$. Последовательность S_1, ..., S_k называется *SLD-выводом* из ***P*** и S_1.

Подстановка θ называется *ответом на запрос G* к программе ***P*,** если

$$<G, \varepsilon> \rightarrow^* <\square, \theta>.$$

Ответ θ содержит требуемые значения целевых переменных из запроса *G*.

Таким образом, *SLD*-резолюция позволяет выводить из одних состояний другие. Мы начинаем с исходного состояния $<G, \varepsilon>$ и пытаемся, исходя из программы ***P***, свести запрос *G* к более простым запросам путем построения *SLD*-вывода. Наша цель – прийти в такое состояние, когда все вопросы решены, и извлечь требуемый ответ.

Система *SLD*-резолюции является *корректной*, т. е. если θ - ответ на запрос *G* вида

$$?- A_1, ..., A_n$$

к программе ***P***, то формула

$$(A_1 \& ... \& A_n)\,\theta$$

является логическим следствием ***P*.** Более того, *SLD*-резолюция *полна* в том смысле, что если $G(X_1, ..., X_k)$ – запрос к программе ***P***, содержащий переменные $X_1, ..., X_k$, и если $G(t_1, ..., t_k)$ – логическое следствие ***P***, то

$$<G, \varepsilon> \rightarrow^* <\square, \theta>,$$

причем $G(t_1, ..., t_k)$ – частный случай $G\theta$, то есть $G(t_1, ..., t_k) = G\theta\theta_1$ для некоторой подстановки θ_1.

7.2 Хорновская логическая программа

Определение. *Хорновским дизъюнктом* называется дизъюнкт, содержащий не более одного позитивного литерала. Дизъюнкт, состоящий только из одного позитивного литерала, называется *фактом*. Дизъюнкт, имеющий позитивный и негативные литералы называется *правилом*.

Логическая или, точнее, *хорновская логическая программа* состоит из набора хорновских дизъюнктов. Структура этих дизъюнктов такова, что, с одной стороны, с их помощью довольно естественно описываются многие задачи, а с другой стороны, они допускают простую процедурную интерпретацию.

Факты в программе описываются в привычном виде, например,

```
P(t1, t2, t3).
```

Наличие в программе факта $P(t_1, ..., t_n)$, содержащего (в своих аргументах) переменные X_1, ..., X_m, означает, что для любых объектов X_1, ..., X_m имеет место $P(t_1, ..., t_n)$.

Правило $A_1 \& ... \& A_n \supset A_0$, где A_i - атомарные формулы, в обозначениях Пролога принято записывать по-иному:

```
A0 :- A1, ..., An.
```

Если в таком правиле встречаются переменные *X1, ..., Xm*, то читается оно следующим образом: для всех объектов *X1, ..., Xm* из *A1, ..., An* следует *A0* (или, что эквивалентно, для всех *X1, ..., Xm*, если верны утверждения *A1, ..., An*, то верно также *A0*).

Запросом к логической программы называется формула вида $A_1 \& ... \& A_n$, где все A_i – атомарные формулы. Следуя синтаксису Пролога, вместо этой формулы мы написали бы

```
?- A1, ..., An.
```

Запрос, не содержащий переменных, читается так: верно ли, что *A1, ... и An*? Если же в атомарных формулах содержатся переменные *X1, ..., Xm*, то его следует читать иначе: для каких объектов *X1, ..., Xm* верно *A1, ... и An*? При этом, разумеется, считается, что машине «известна» только та информация, которая представлена в программе (и, быть может, свойства некоторых встроенных предикатов и функций, таких, как < или +), и ответ на запрос должен логически вытекать из этой информации.

Таким образом, логические программы имеют простую и естественную семантику: запрос

```
?- A1, ..., An.
```

(с переменными *X1, ..., Xm*) к программе *P* понимается как требование вычислить все значения переменных *X1, ..., Xm*, при которых утверждения *A1*, ... и *An* логически следуют из утверждений, содержащихся в *P*.

Пример простейшей программы на Прологе выглядит так:

```
member(X, [X|_]).
member(X, [_|T]):-
           member(X,T).
```

Эту программу можно прочитать так: «*X* является членом списка, если он совпадает с головой списка, или является членом хвоста списка». В этой программе объявлен один предикат – `member`.

Обычно Пролог–система работает в форме диалога с пользователем. Утверждение, которое требуется доказать, вводится с клавиатуры.

Для данной программы можно задавать различные вопросы Пролог системе. Вызов

```
?- member([2,[1,2,3]).
```

означает вопрос «2 является элементом списка [1,2,3]?».
Пролог ответит
Yes

На более сложный вопрос

```
?- member([X,[1,2,3]).
```

(интерпретируется как «для какого X верно, что он является членом списка?») последуют ответы

```
X=1;
X=2;
X=3
```

Пример из 6.10 запишется в виде следующей программы на Прологе.

```
m(X):-
          f(X,Y).
s(Y,W):-
          f(X,Y),f(X,W).
b(X,Y):-
          s(X,Y),m(X).
f(john,harry).
f(john,sid).
f(sid,liz).

?- b(Z,harry).
```

Опишем более систематично процесс программирования на языке Пролог.

Решение задач. Конечной целью составления компьютерной программы является создание инструмента, предназначенного для решения задачи из некоторой прикладной области. *Прикладная область* - это некоторая абстрактная часть мира или область знаний. *Структура* прикладной области состоит из значимых для этой области сущностей, функций и отношений. В удачной компьютерной программе структура прикладной области моделируется таким образом, что поведение этой программы во время выполнения будет отражать какие-то существенные аспекты данной структуры. К примеру, отношение, существующее между некоторыми сущностями прикладной области, должно, по аналогии, соблюдаться и для обозначений, которые представляют эти сущности в программе, моделирующей эту область.

Программирование. Процесс составления программы на Прологе в основном сходен с процессом построения теории в логике предикатов.

- Программист анализирует значимые сущности, функции и отношения из прикладной области и выбирает обозначения для них.
- Программист семантически определяет каждую значимую функцию и каждое значимое отношение. Для отношений указывается, какие конкретные их реализации дают *истину*, а какие - *ложь*.
- Программист аксиоматически определяет каждое отношение при помощи правил и фактов Пролога. Аксиоматическое определение отношения будет удачным, если оно отразит смысл семантического определения. Множеством аксиоматических определений всех значимых для заданной предметной области отношений является программа, моделирующая структуру этой области.
- Для выполнения запросов к множеству правил и фактов программист или пользователь применяет интерпретатор языка Пролог. Совокупность, состоящую из запроса, множества правил и фактов программы и интерпретатора языка, можно рассматривать как алгоритм решения задач из прикладной области. При этом запрос, правила и факты программы представляют собой начальные формулы алгоритма, а интерпретатор содержит правила преобразования этих формул. Интерпретатор играет роль активной силы, которая выполняет выводы из правил программы и тем самым *реализует* или *развертывает* отношения определенные правилами программы.

Р. Ковальский описывает сущность логического программирования фразой:

Алгоритм = Логика + Управление.

Подобно тому, как Лисп скрыл от программиста устройство памяти компьютера, Пролог позволил ему не заботиться (без необходимости) о потоке управления в программе.

7.3 Оценка языка Пролог

При написании данного пункта, мы следуем мнению Н. Н. Непейводы[6].

Язык Пролог рекламируется как язык логического программирования. Программа состоит из множества хорновских дизъюнктов, записываемых в виде

$$Q(\mathbf{t}) \text{ :- } P_1(\mathbf{t_1}),\ P_2(\mathbf{t_2}),\ \ldots,\ P_n(\mathbf{t_n}).$$

(жирным шрифтом выделены кортежи: на самом деле предикаты могут иметь любое число аргументов).

Кроме того, в программе имеется цель вида

$$?-\ R_1,\ \ldots,\ R_k.$$

Каждый шаг программы состоит в преобразовании цели путем ее унификации с одним из дизъюнктов. Программа считается успешно завершенной, если в некоторый момент из цели исчезают все предикаты. Программа может зафиксировать неудачу, если один из предикатов цели ни с одним из дизъюнктов программы не унифицируется. Естественно, что может быть и промежуточный, но гораздо чаще встречающийся случай: программа не может зафиксировать неудачу, а просто зацикливается либо переполняется из-за неограниченного удлинения выражений.

Эта схема могла быть реализована многими способами. Поскольку Пролог появился в самом начале 70-ых годов, был выбран способ, тогда находившийся на уровне, но сейчас уже безнадежно морально устаревший[1]. Выбирается всегда первый член целевого дизъюнкта и первый из унифицируемых с ним дизъюнктов программы.

Здесь возникла сложность, которая была удачно разрешена и составила одно из важнейших достижений Пролога. Взяв первого кандидата, мы можем через несколько шагов зайти в тупик, а решение было совсем рядом: надо было взять следующего. Тут работает механизм возвратов (backtracking). Если фиксируется неудача, мы возвращаемся к

[1] Впрочем, уже тогда можно было бы чуть дальше глянуть на уже имевшиеся достижения информатики, но, как правило, больше одного удачного нововведения ни в одной принципиально новой системе не делается.

первой точке, где было несколько кандидатов на унификацию, и подставляем следующий из возможных дизъюнктов. Этот механизм явился красивой и экономичной с точки зрения представления программ альтернативой явному выписыванию условных операторов. Но, конечно же, с точки зрения исполнения программ он может безнадежно проигрывать в эффективности.

Правило «брать первого кандидата из не отвергнутых ранее» обладает и другими особенностями. Во-первых, логика перестает быть классической, поскольку тривиально истинный дизъюнкт вида

```
P(X) :- P(X).
```

при помещении в программу вполне может привести к ее зацикливанию (если он применится однажды, то он будет применяться бесконечно). Во-вторых (и это уже большой плюс), появляется возможность выражать циклы и индукцию при помощи правил типа

```
A(N) :- N1 is N-1,
        A(N1),
        B.
```

Далее Пролог, некоторое время просуществовав в университетской среде, неожиданно получил громадную рекламу в связи с тем, что японцы объявили его внутренним языком своего проекта ЭВМ пятого поколения. В итоге была набрана критическая масса людей, которые знают Пролог, получили под него ассигнования и больше ничего знать не хотят. Более того, сам термин «логическое программирование» сейчас понимается как программирование на Прологе.

8 Функциональный взгляд на вычисления

На протяжении последних 400 лет, центральным понятием математики является понятие *функции*. Математические функции выражают связь между параметрами (входом) и результатом (выходом) некоторого процесса. Так как вычисление это тоже процесс, имеющий вход и выход, функция – вполне подходящее средство задания вычислений. Программа есть формальное описание конкретного вычисления. Если мы игнорируем процедурные детали («как» вычисления делаются) и только принимаем во внимание результат («что» вычисления сделали), то мы можем рассматривать программу как «черный ящик», который получает какой-то вход и возвращает какой-то выход. Другими словами, программа может быть рассмотрена как математическая функция.

Мы можем думать о программах, процедурах и функциях в языке программирования, как если бы они представляют математическое понятие функции:

- в случае программы, x представляет вход и y представляет выход;
- в случае процедуры или функции, x представляет параметры, а y представляет возвращаемую величину.

В любом случае мы можем ссылаться на x как «вход» и на y как «выход». Поэтому функциональный взгляд на программирование не делает различие между программой, процедурой или функцией. Но он, однако, всегда различает входную и выходную величины.

В языке программирования мы также должны различать определение функции и применение функции: первое описывает, как функция будет вычисляться, используя формальные параметры, в то время как функциональное применение есть вызов описанной функции, использующий фактические параметры или аргументы.

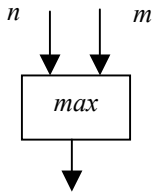
С абстрактной точки зрения мы можем рассматривать функции как «черные ящики». Рассматривая функции как черный ящик, мы можем использовать его как конструктивный блок, и с помощью объединения (когда выход одного ящика подается на вход другого) таких ящиков можно порождать более сложные операции. Такой процесс «объединения» ящиков называется *функциональной композицией*.

Пусть дана математическая функция с действительными числами *max*: $\mathbf{R}\times\mathbf{R} \to \mathbf{R}$, определяющая наибольшее значение из двух чисел:

$max(m,n) = m$, если $m>n$,

$= n$ в противном случае.

Ее представление в виде черного ящика очень просто:



Можно использовать *max* как блок для более сложной функции. Предположим, требуется построить функцию, которая находила бы максимум не из двух чисел, а из трех. Эту новую функцию (назовем ее *max*3) определим следующим образом:

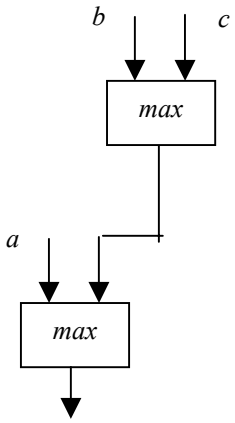
$max3\ (a,b,c) = a$, если $a\geq b$ и $a>c$ или $a\geq c$ и $a>b$,

b, если $b\geq a$ и $b>c$ или $b\geq c$ и $b>a$,

c, если $c\geq a$ и $c>b$ или $c\geq b$ и $c>a$,

a в противном случае.

Это довольно неудобное определение. Гораздо более элегантный способ определения функции *max*3 состоит в использовании уже определенной функции *max*:



Это можно записать следующим образом:
*max*3(*a,b,c*) = *max*(*a, max*(*b,c*)).

Поскольку функция обеспечивает детерминированное отображение тройки чисел в число, то ее можно рассматривать как черный ящик с тремя входами и одним выходом.

Таким образом, задав множество предварительно определенных черных ящиков–функций, называемых *встроенными функциями* или *примитивами***,** для выполнения простых операций, подобных базовым арифметическим, можно построить новые функции, т.е. новые черные ящики для выполнения более сложных операций с помощью этих примитивов. Далее, эти новые функции можно использовать как блоки для построения еще более сложных функций и т.д.

Программа на функциональном языке состоит из множества определений функций и выражения, чья величина рассматривается как результат программы. Математической моделью функциональных языков является ламбда–исчисление. Примерами функциональных языков являются Clean, Haskell и др. Многие другие языки, такие как Лисп, имеют чисто функциональное подмножество, но также содержат нефункциональные конструкции.

Чтобы понять решение на императивном языке, мы должны понять, что машина будет делать при выполнении каждого оператора программы. При функциональном решении, с другой стороны, мы не должны думать о том, как программа будет выполняться на компьютере; отсутствует всякое упоминание об изменяемом состоянии программы или о последовательном выполнении инструкций. Функциональное решение является фактически формулировкой самой задачи, а не рецептом ее решения, и именно в этом смысле мы говорим о функциональной программе как о спецификации того, что нужно сделать вместо последовательности инструкций, описывающих, как это сделать. Таким образом, функциональное программирование является одним из видов декларативного программирования.

Функциональное программирование, по сравнению с другими парадигмами программирования, предлагает наиболее точный изоморфизм между постановкой задачей и программой.

В функциональных языках все вычисления выполняются через оценку выражений, чтобы выдать какие-то величины.

В математике переменные всегда представляют какие-то конкретные величины, в то время как в императивном программировании переменные ссылаются на конкретное место в памяти и так же на величину. В математике нет понятия «место в памяти», так что выражение $x = x+1$ не имеет смысла. Функциональный взгляд на программирование поэтому должен уничтожить понятие переменной, иное чем имя величины. Точно также исчезает присваивание как допустимый оператор.

Этот взгляд на функциональное программирование приводит к понятию «*чистое функциональное программирование*». Большинство функциональных языков оставляет некоторое понятие переменной и присваивания, и поэтому «не чисты», но все же возможно

программировать эффективно, используя чистый подход. Haskell является чистым функциональным языком.

Одним из следствий отсутствия переменных и присваивания, является невозможность циклов: цикл должен иметь управляющую переменную, которая должна менять свое значение во время выполнения цикла, но это невозможно без переменных и присваиваний. Вместо этого мы должны использовать рекурсию.

Фундаментальное свойство математических функций, которое дает нам возможность собрать воедино черные ящики, – это *функциональность* (*прозрачность по ссылкам*).

Принцип *прозрачности по ссылкам* утверждает: «Значение целого не меняется, когда какая-то часть целого заменяется на равную часть». Выражение E является прозрачным по ссылкам, если любое подвыражение и его величина (как результат его вычисления) могут быть взаимно заменены без изменения величины E.

Это означает, что каждое выражение определяет единственную величину, которую нельзя изменить ни путем ее вычисления, ни предоставлением различным частям программы возможности совместно использовать это выражение. Вычисление выражения просто изменяет форму выражения, но не изменяет его величину. Все ссылки на некоторую величину эквивалентны самой этой величине, и тот факт, что на выражение можно ссылаться из другой части программы, никак не влияет на величину этого выражения.

Из-за отсутствия переменных и присваиваний в языке отсутствует понятие состояния функции. В императивном программировании два главных фактора действуют на внутреннее состояние процедуры:

- предыдущие вычисления (предшествующие вызову процедуры), включающие побочный эффект от предыдущих вызовов самой процедуры;
- порядок оценки параметров при вызове.

В императивных языках (например, Паскаль) функции могут ссылаться на глобальные данные и разрешается применять деструктивное (разрушающее) присваивание, что может привести к изменению значения функции при повторном ее вызове. Такие динамические изменения в величине данных часто именуются *побочными эффектами*. Благодаря им значение функции может изменяться, даже если ее аргументы и остаются без изменения всякий раз, когда к ней обращаются.

Пример нефункциональности Паскаля:

```
var flag: boolean;

    function f (n:integer):integer;
    begin
        if flag then f:=n
                  else f:= 2*n;
        flag:=not flag
    end;

    begin
       flag:=true;
       writeln(f(1)+f(2));     {будет напечатано 5}
       writeln(f(2)+f(1));     {будет напечатано 4}
    end.
```

Функция f не является математической («чистой») функцией. Операции, подобные этой, в математике не разрешены, поскольку математические рассуждения базируются на идее равенства и возможности замены одного выражения другим, означающим то же самое, т.е. определяющим ту же величину.

В функциональном программировании, однако, значение функции зависит только от значений ее параметров, но не от предыдущих вычислений, включая и вызовы самой функции. Величина любой функции также не может зависеть от порядка вычисления ее

параметров (тем самым функциональные языки становятся подходящими для параллельных приложений).

Прозрачная по ссылкам функция без параметров должна всегда выдавать тот же самый результат, и поэтому ничем не отличается от константы. Более того, различные вызовы одной и той же функции с теми же аргументами будут всегда возвращать один и тот же результат.

В функциональном программировании имеется возможность оперировать функциями всевозможными способами, без ограничений. В частности функция сама по себе может быть рассматриваться как структура данных, которая может быть аргументом другой функции и которая может возвращаться как результат вычисления функции.

Функциональные языки все в большей и большей степени используются в образовании, потому что они способствуют выражать концепции и вычислительные структуры программирования на высоком уровне абстракции. Многие зарубежные университеты в области компьютерных наук используют функциональное программирование в качестве основы курса для тех, кто углубленно изучает программирование, и на нескольких факультетах в качестве первого языка программирования студенты изучают функциональный язык.

9 Ламбда–исчисление

Желтая река течет тысячи миль на север…
Затем поворачивает на восток и течет непрерывно,
Не важно, как она изгибается и поворачивается,
Её воды выходят из источника на горе Кунь–Лунь.
Железная флейта

9.1 Ламбда–исчисление как формальная система

9.1.1 Значение ламбда–исчисления

Ламбда–исчисление было изобретено Алонсом Чёрчем около 1930 г. Чёрч первоначально строил λ–исчисление как часть общей системы функций, которая должна стать основанием математики. Но из–за найденных парадоксов эта система оказалась противоречивой. Книга Чёрча [12] содержит непротиворечивую подтеорию его первоначальной системы, имеющую дело только с функциональной частью. Эта теория и есть λ–исчисление.

Алонсо Чёрч

Ламбда–исчисление – это безтиповая теория, рассматривающая функции как *правила*, а не как графики. В противоположность подходу Дирихле (вводимому функции как множество пар, состоящих из аргумента и значения) более старое понятие определяет функцию как процесс перехода от аргумента к значению. С первой точки зрения x^2–4 и (x+2)(x–2) – разные обозначения одной и той же функции; со второй точки зрения это разные функции.

Что значит «функция $5x^3$+2»? Если кто–то хочет быть точным, он вводит по этому поводу функциональный символ, например f, и говорит: «функция f: $\mathbf{R} \to \mathbf{R}$, определенная соотношением $f(x) = 5x^3+2$». При этом, очевидно, что переменную x можно здесь, не меняя смысла, заменить на другую переменную y. Ламбда–запись устраняет произвольность в выборе f в качестве функционального символа. Она предлагает вместо f выражение «$\lambda x.5x^3+2$».

Кроме того, обычная запись $f(x)$ может обозначать как имя функции f, так и вызов функции с аргументом x. Для более строгого подхода это необходимо различать. В ламбда–обозначениях вызов функции с аргументом x выглядит как $(\lambda x.\ 5x^3+2)x$.

Функции как правила рассматриваются в полной общности. Например, мы можем считать, что функции заданы определениями на обычном русском языке и применяются к аргументам также описанным по–русски. Также мы можем рассматривать функции заданными программами и применяемые к другим программам. В обоих случаях перед нами *безтиповая структура*, где объекты изучения являются одновременно и функциями и аргументами. Это отправная точка безтипового λ–исчисления. В частности, функция может применяться к самой себе.

Ламбда–исчисление представляет класс (частичных) функций (λ–определимые функции), который в точности характеризует неформальное понятие эффективной вычислимости. Другими словами, λ–исчисления, наряду с другими подходами формализует понятие алгоритма [1, с. 143–146].

Ламбда–исчисление стало объектом особенно пристального внимания в информатике после того, как выяснилось, что оно представляет собой удобную теоретическую модель современного функционального программирования [8].

Большинство функциональных языков программирования используют λ–исчисление в качестве промежуточного кода, в который можно транслировать исходную программу. Функциональные языки «улучшают» нотацию λ–исчисления в прагматическом смысле, но при этом, в какой–то мере, теряется элегантность и простота последнего.

Изучение и понимание многих сложных ситуаций в программировании, например, таких, как автоаппликативность (самоприменимость) или авторепликативность (самовоспроизводство), сильно облегчается, если уже имеется опыт работы в λ–исчислении, где выделены в чистом виде основные идеи и трудности.

Язык ламбда–исчисления является сейчас одним из важнейших выразительных средств в логике, информатике, математической лингвистике, искусственном интеллекте и когнитивной науке.

9.1.2 Синтаксис и семантика ламбда–исчисления

Ламбда–исчисление есть язык для определения функций. Выражения языка называются λ–*выражениями* и каждое такое выражение обозначает функцию. Далее мы рассмотрим, как функции могут представлять различные структуры данных: числа, списки и т. д. Для некоторых λ–выражений мы будем использовать имена (или сокращенные обозначения), они будут записываться полужирным шрифтом.

Определение λ–выражений (λ–термов)

Имеется три вида λ–выражений:

- *Переменные*: x, y и т. д.
- *Функциональная аппликация*: если M и N есть произвольные λ–термы, то можно построить новый λ–терм (MN) (обозначающий применение, или *аппликацию*, оператора M к аргументу N). Например, если (<m>, <n>) обозначает функцию, представляющую пару чисел m и n и ***sum*** обозначает функцию сложения в λ–исчислении, то аппликация (***sum*** (<m>, <n>)) обозначает <m+n> (т. е. ламбда–терм, представляющий число m+n).
- *Абстракция*: По любой переменной x и любому λ–терму M можно построить новый λ–терм ($\lambda x.M$) (обозначающий функцию от x, определяемую λ–термом M). Такая конструкция называется λ–*абстракция***.** Например, $\lambda x.$ ***sum*** (x, <1>) обозначает функцию от x, которая увеличивает аргумент на 1.

Тождественная функция представляется λ–термом ($\lambda x.\ x$). Аппликация ($\lambda x.\ x$) E дает E.

Еще пример. Пусть λ–выражения <0>, <1>, <2>, … представляют числа 0, 1, 2, …, соответственно; и ***add*** есть λ–выражение, обозначающее функцию, удовлетворяющее правилу:

$$((\textbf{\textit{add}}\ \langle m\rangle)\ \langle n\rangle) = \langle m+n\rangle.$$

Тогда ($\lambda x.$ ((***add*** <1>) x)) есть λ–выражение, обозначающее функцию, что преобразует <n> в <n+1>, и ($\lambda x.$ ($\lambda y.$ ((***add*** x) y))) есть λ–выражение, обозначающее функцию, которая преобразует <m> в функцию ($\lambda y.$ ((***add*** <m>) y)).

Символ x после λ называется *связанной переменной* абстракции и соответствует понятию формального параметра в традиционной процедуре или функции. Выражение справа от точки называется *телом абстракции*, и, подобно коду традиционной процедуры или функции, оно описывает, что нужно сделать с параметром, поступившим на вход функции. Мы читаем символ λ как «функция от» и точку (.) как «которая возвращает».

Чтобы уменьшить число применяемых скобок будем использовать следующие соглашения.

- Операция аппликации лево–ассоциативна, т. е. $E_1\ E_2\ \ldots\ E_n$ означает

((… (E_1 E_2)…)E_n). Например, E_1 E_2 E_3 обозначает ((E_1 E_2) E_3).

- $\lambda V.\ E_1\ E_2\ \ldots\ E_n$ обозначает ($\lambda V.$ (E_1 E_2 … E_n)). Область действия λV распространяется направо так далеко как это возможно.
- Операция абстракции является право–ассоциативной, т. е. $\lambda V_1.\ \lambda V_2.\ \ldots\ \lambda V_n.\ E$ обозначает ($\lambda V_1.$ ($\lambda V_2.$ (… ($\lambda V_n.\ E$) …))). Запись $\lambda V_1.\ \lambda V_2.\ \ldots\ \lambda V_n.\ E$ иногда еще более сокращают $\lambda V_1\ V_2\ \ldots\ V_n.\ E$. Так, например, $\lambda x\ y\ z.\ E$ обозначает ($\lambda x.(\lambda y.(\lambda z.E))$).

9.1.3 Вычисление ламбда–выражений

Переменная, расположенная не на месте связанной переменной, может быть *связанной* или *свободной*, что определяется с помощью следующих правил:

1. Переменная x оказывается свободной в выражении x.
2. Все x, имеющиеся в $\lambda x.M$, являются связанными. Если кроме x в $\lambda x.M$ есть переменная y, то последняя будет свободной или связанной в зависимости от того, свободно она или связанна в M.
3. Переменная встречающаяся в термах M или N выражения (MN) будет связанной или свободной в общем терме в зависимости от того, свободна она или связанна в M или N. Свободные (связанные) переменные – это переменные, которые, по крайней мере, один раз появляются в терме в свободном (связанном) виде.

Нам понадобится следующее определение *подстановки* терма в другой терм вместо свободного вхождения переменной. Для любых λ–термов M, N и переменной x через $[N/x]M$ обозначим результат подстановки N вместо каждого свободного вхождения x в M и замены всех λy в M таким образом, чтобы свободные переменные из N не стали связанными в $[N/x]M$. Мы употребляем запись $M \equiv N$ для обозначения того, что термы M и N совпадают.

Подстановка

a) $[N/x]x \equiv N$;
b) $[N/x]y \equiv y$, если переменная y не совпадает с x;
c) $[N/x]\ (PQ) \equiv ([N/x]P\ [N/x]Q)$;
d) $[N/x]\ (\lambda x.P) \equiv \lambda x.P$;
e) $[N/x]\ (\lambda y.P) \equiv \lambda y.[N/x]P$, если y не имеет свободных вхождений в N и x имеет свободное вхождение в P;
f) $[N/x](\lambda y.P) \equiv \lambda z.[N/x]([z/y]P)$, если y имеет свободное вхождение в N и x имеет свободное вхождение в P и z – любая переменная, не имеющая свободных вхождений в N.

Следующие примеры пояснят суть определения. Пусть $M \equiv \lambda y.yx$.

Если $N \equiv vx$, то $[(vx)/x](\lambda y.yx) \equiv \lambda y.\ [(vx)/x](yx)$ согласно (e)
$\equiv \lambda y.\ y(vx)$ согласно (a).

Если $N \equiv yx$, то $[(yx)/x](\lambda y.yx) \equiv \lambda z.\ [(yx)/x](zx)$ согласно (f)
$\equiv \lambda z.z(yx)$ согласно (a).

Введенное понятие подстановки требует должной мотивации. Для этого необходимо снова вспомнить о коллизии переменных при неаккуратной подстановке (см. 2.3.4).

Если бы пункт (f) в определении подстановки был опущен, то мы столкнулись бы со следующим нежелательным явлением. Хотя $\lambda v.x$ и $\lambda y.x$ обозначают одну и ту же функцию (константу, чье значение всегда есть x), после подстановки v вместо x они стали бы обозначать разные функции: $[v/x](\lambda y.x) \equiv \lambda y.v$, $[v/x](\lambda v.x) \equiv \lambda v.v$.

Мы рассмотрели, как λ–нотация может быть использована для представления функциональных выражений и сейчас готовы к тому, чтобы определить *правила конверсии* λ–исчисления, которые описывают, как вычислять выражение, т. е. как получать ко-

нечное значение выражения из его первоначального вида. Правила конверсии, описанные ниже, являются достаточно общими, так, что, например, когда они применяются к λ–терму, представляющему арифметическое выражение, то они моделируют вычисление этого выражения.

Правила конверсии

- **α–конверсия.** Любая абстракция вида $\lambda V.\ E$ может быть конвертирована к терму $\lambda V'.\ [V'/V]E$. Мы переименовываем связные переменные так, чтобы избежать коллизии переменных.
- **β–конверсия.** Любая аппликация вида $(\lambda V.\ E_1)\ E_2$ конвертируется к терму $[E_2/V]E_1$.
- **η–конверсия.** Любая абстракция вида $\lambda V.\ (E\ V)$, в которой V не имеет свободных вхождений в терме E может быть конвертировано к E.

Если какой-то λ–терм E_1 α–конвертируется к терму E_2, то это обозначается как $E_1 \rightarrow_\alpha E_2$. Аналогично определяются обозначения $\rightarrow_\beta$ и $\rightarrow_\eta$. Наиболее важным видом конверсии является β–конверсия; она единственная может использоваться для моделирования произвольного вычислительного механизма. α–конверсия применяется для технических преобразований связных переменных и η–конверсия выражает тот факт, что две функции, дающие один и тот же результат для любых аргументов, естественно считаются равными.

Обсудим более подробно правила конверсии. Ламбда–выражение, к которому может быть применено правило α–конверсии, называется α–*редексом*. «Редекс» есть абревиатура для «редуцируемое выражение». Правило α–конверсии говорит, что любая связанная переменная может быть корректно переименована. Например, $\lambda x.x \rightarrow_\alpha \lambda y.y$.

Ламбда–выражение, к которому может быть применено правило β–конверсии, называется β–*редексом*. Правило β–конверсии подобно вычислению вызова функции в языке программирования: тело E_1 функции $\lambda V.\ E_1$ вычисляется в окружении (в контексте), в котором «формальный параметр» V заменяется на «фактический параметр» E_2.

Примеры:

$$(\lambda x.\ f\ x)\ E \rightarrow_\beta f\ E$$
$$(\lambda x.\ (\lambda y.\ (\mathbf{add}\ x\ y))) <3> \rightarrow_\beta \lambda y.\ \mathbf{add}\ <3>\ y$$
$$(\lambda y.\ \mathbf{add}\ <3>\ y)\ <4> \rightarrow_\beta \mathbf{add}\ <3>\ <4>$$

Ламбда–выражение, к которому может быть применено правило η–конверсии, называется η–*редексом*. Правило η–конверсии выражает следующее свойство (называемое *экстенсиональностью*): две функции являются равными, если они дают одинаковый результат когда применяются к одинаковым аргументам. Действительно, $\lambda V.\ (E\ V)$ обозначает функцию, которая возвращает $[E'/V]\ (E\ V)$ когда применяется к аргументу E'. Если V не является свободной в E, то $[E'/V]\ (E\ V) = (E\ E')$. Так как термы $\lambda V.\ (E\ V)$ и E оба возвращают один и тот же результат $(E\ E')$, когда применяются к одинаковым аргументам, то, следовательно, они обозначают одну и ту же функцию.

Примеры:

$$\lambda x.\ \mathbf{add}\ x \rightarrow_\eta \mathbf{add}$$
$$\lambda y.\ \mathbf{add}\ x\ y \rightarrow_\eta \mathbf{add}\ x$$

Но следующая конверсия

$$\lambda x.\ \mathbf{add}\ x\ x \rightarrow_\eta \mathbf{add}\ x$$

не имеет место, поскольку переменная *x* свободна в ***add*** *x*.

Более общим понятием, по сравнению с конверсией, является понятие *обобщенной конверсии* (по причинам, которые станут понятным позже, она также называется *одношаговой редукцией*). Терм *M* обобщенно α–конвертируется (β–конвертируется, η–конвертируется) к терму *N*, если последний получается из *M* в результате конверсии какого-то подтерма в *M*, являющегося α–редексом (β–редексом, η–редексом, соответственно). Обобщенная конверсии обозначается также как и просто конверсия: M $\rightarrow_\alpha$ N (аналогично и для других видов обобщенной конверсии).

В следующих примерах конвертируемые редексы подчеркнуты.

$$\underline{((\lambda x.\ (\lambda y.\ (\textbf{\textit{add}}\ x\ y)))\ \langle 3\rangle)}\ \langle 4\rangle \rightarrow_\beta\ (\lambda y.\ \textbf{\textit{add}}\ \langle 3\rangle\ y)\ \langle 4\rangle$$
$$\underline{(\lambda y.\ \textbf{\textit{add}}\ \langle 3\rangle y)\ \langle 4\rangle} \rightarrow_\beta\ \textbf{\textit{add}}\ \langle 3\rangle\ \langle 4\rangle$$

Отметим, что в первой конверсии сам терм ((λ*x*. (λ*y*. (***add*** *x y*))) <3>)<4> не является редексом. Мы будем иногда писать последовательность конверсий, подобных двум предыдущим как:

$$\underline{((\lambda x.\ (\lambda y.\ (\textbf{\textit{add}}\ x\ y)))\ \langle 3\rangle)}\ \langle 4\rangle \rightarrow_\beta\ (\lambda y.\ \textbf{\textit{add}}\ \langle 3\rangle\ y)\ \langle 4\rangle \rightarrow_\beta\ \textbf{\textit{add}}\ \langle 3\rangle\ \langle 4\rangle$$

В следующих двух примерах исходное исходное выражение одно и то же, но последовательности конвертируемых редексов различны (используемые редексы подчеркнуты):

1) $\underline{(\lambda f.\ \lambda x.\ f\ \langle 3\rangle\ x)\ (\lambda y.\ \lambda x.\textbf{\textit{add}}\ x\ y)}\ \langle 0\rangle$
$\rightarrow_\beta\ \underline{(\lambda x.(\lambda y.\ \lambda x\ .\ \textbf{\textit{add}}\ x\ y)\ \langle 3\rangle\ x)\ \langle 0\rangle}$
$\rightarrow_\eta\ \underline{(\lambda z.(\lambda y.\ \lambda x\ .\ \textbf{\textit{add}}\ x\ y)\ \langle 3\rangle\ z)\ \langle 0\rangle}$ – выбрали один из двух возможных редексов
$\rightarrow_\beta\ \underline{(\lambda y.\lambda z.\ \textbf{\textit{add}}\ z\ y)\ \langle 3\rangle}\ \langle 0\rangle$
$\rightarrow_\beta\ \underline{(\lambda z.\ \textbf{\textit{add}}\ z\ \langle 3\rangle)\ \langle 0\rangle}$
$\rightarrow_\beta\ \textbf{\textit{add}}\ \langle 0\rangle\ \langle 3\rangle$

2) $\underline{(\lambda f.\ \lambda x\ .f\ \langle 3\rangle\ x)\ (\lambda y.\ \lambda x.\ \textbf{\textit{add}}\ x\ y)}\ \langle 0\rangle$
$\rightarrow_\beta\ (\lambda x.\ \underline{(\lambda y.\ \lambda x.\ \textbf{\textit{add}}\ x\ y)\ \langle 3\rangle}\ x)\ \langle 0\rangle$ – выбрали один из двух возможных редексов
$\rightarrow_\eta\ \underline{(\lambda x.(\lambda x.\ \textbf{\textit{add}}\ x\ \langle 3\rangle)x\ )\ \langle 0\rangle}$
$\rightarrow_\beta\ \underline{(\lambda z.(\lambda x.\ \textbf{\textit{add}}\ x\ \langle 3\rangle)z\ )\ \langle 0\rangle}$ – снова делаем произвольный выбор
$\rightarrow_\beta\ \underline{(\lambda z.\ \textbf{\textit{add}}\ z\ \langle 3\rangle)\langle 0\rangle}$
$\rightarrow_\beta\ \textbf{\textit{add}}\ \langle 0\rangle\ \langle 3\rangle$

Заметим, что в данном случае независимо от выбора редексов мы пришли к одинаковому результату.

9.1.4 Нормальные формы

Если λ–выражение *E* получается из λ–выражения *E′* с помощью последовательности обобщенных конверсий, то естественно считать, что *E′* получается в результате «вычисления» *E*, которое более точно выражается через понятие *редукции*.

Определение отношения редукции

Пусть *E* и *E′* есть λ–выражения. Говорят, что терм *E* редуцируется к терму *E′* (и обозначают как $E \rightarrow E'$), если $E \equiv E'$ или существуют выражения $E_1, E_2, \ldots, E_n$ такие что:

1. $E \equiv E_1$
2. $E_n \equiv E'$
3. Для каждого i выполнено одно из трех отношений $E_i \rightarrow_\alpha E_{i+1}$, $E_i \rightarrow_\beta E_{i+1}$ или $E_i \rightarrow_\eta E_{i+1}$.

Последний пример предыдущего пункта говорит, что

$$(\lambda f.\ \lambda x\ .f\ \langle 3\rangle\ x)\ (\lambda y.\ \lambda x.\ \textbf{\textit{add}}\ x\ y)\ \ \langle 0\rangle \rightarrow\ \textbf{\textit{add}}\ \ \langle 0\rangle\ \langle 3\rangle$$

Три правила конверсии сохраняют значения λ–выражений, т. е. если терм E редуцируется к терму E', то выражения E и E' обозначают одну и ту же функцию. Мы определим понятие равенства λ–выражений.

Пусть E и E' есть λ–выражения. Говорят, что терм E *равен* терму E' (обозначают как $E = E'$), если $E \rightarrow E'$ или $E' \rightarrow E$. Мы требуем также, чтобы введенное отношение равенства для λ–выражений обладало свойством транзитивности. В частности, если $E_1 \rightarrow E'$ и $E_2 \rightarrow E'$, то считается также $E_1 = E_2$.

Говорят, что λ–выражение находится в *нормальной форме*, если к нему нельзя применить никакое правило конверсии. Другими словами, λ–выражение – в нормальной форме, если оно не содержит редексов. Нормальная форма, таким образом, соответствует понятию конца вычислений в традиционном программировании. Отсюда немедленно вытекает наивная схема вычислений:

```
while существует хотя бы один редекс
do преобразовывать один из редексов
end
```

(выражение теперь в нормальной форме).

Различные варианты конверсии в процессе редукции могут приводить к принципиально различным последствиям.

Пример:

$$(\lambda x.\lambda y.y)\ (\underline{(\lambda z.\ z\ z)\ (\lambda z.z\ z)})$$
$$\rightarrow (\lambda x.\lambda y.y)\ (\underline{(\lambda z.z\ z)\ (\lambda z.z\ z)})$$
$$\rightarrow (\lambda x.\lambda y.y)\ (\underline{(\lambda z.z\ z)\ (\lambda z.z\ z)})$$
$$\rightarrow \ldots$$

(бесконечный процесс редукции)

$$\underline{(\lambda x.\lambda y.y)\ ((\lambda z.\ z\ z)\ (\lambda z.z\ z))}$$
$$\rightarrow \lambda y.y$$

(редукция закончилась)

Порядок редукций (стратегия выбора редексов)

Самым левым редексом называется редекс, первый символ которого текстуально расположен в λ–выражении левее всех остальных редексов. (Аналогично определяется *самый правый редекс*.)

Самым внешним редексом называется редекс, который не содержится внутри никакого другого редекса.

Самым внутренним редексом называется редекс, не содержащий других редексов.

В контексте функциональных языков и λ–исчисления существуют два важных порядка редукций [8].

Аппликативный порядок редукций (АПР), который предписывает вначале преобразовывать самый левый из самых внутренних редексов.

Нормальный порядок редукций (НПР), который предписывает вначале преобразовывать самый левый из самых внешних редексов.

В терме $(\lambda x.\lambda y.y)\ (\underline{(\lambda z.zz)\ (\lambda z.z\ z)})$ подчеркнут самый левый из самых внутренних редексов – если его последовательно конвертировать, то редукция никогда не закончится.

В терме $\underline{(\lambda x.\lambda y.y)\ ((\lambda z\ .z\ z)\ (\lambda z\ .zz))}$ подчеркнут самый левый из самых внешних редексов – редукция в этом случае заканчивается за один шаг.

Функция $\lambda x.\lambda y.y$ – это классический пример функции, которая отбрасывает свой аргумент. НПР в таких случаях эффективно откладывает вычисление любых редексов внутри выражения аргумента до тех пор, пока это возможно, в расчете на то, что такое вычисление может оказаться ненужным.

В функциональных языках стратегии НПР соответствуют так называемые *ленивые вычисления*. «Не делай ничего, пока это не потребуется» – механизм вызова по необходимости (все аргументы передаются функции в не вычисленном виде и вычисляются только тогда, когда в них возникает необходимость внутри тела функции). Haskell – один из языков с ленивыми вычислениями.

Стратегия АПР приводит к *энергичным вычислениям*. «Делай все, что можешь»; другими словами, не надо заботиться о том, пригодится ли, в конечном счете, полученный результат. Таким образом, реализуется механизм вызова по значению (значение аргумента передается в тело функции). В языке Лисп реализуются, как правило, энергичные вычисления.

Теорема 14 (Теорема Чёрча-Россера о свойстве ромба) [1, с.78]**.** Если $E \to E_1$ и $E \to E_2$, то существует терм E' такой, что $E_1 \to E'$ и $E_2 \to E'$.

Следствие 1. Если $E_1 = E_2$, то существует терм E' такой, что $E_1 \to E'$ и $E_2 \to E'$.

Следствие 2. Если выражение E может быть приведено двумя различными способами к двум нормальным формам, то эти нормальные формы совпадают или могут быть получены одна из другой с помощью замены связанных переменных.

Теорема 15 (Теорема стандартизации) [1, с. 298–303]. Если выражение E имеет нормальную форму, то НПР гарантирует достижение этой нормальной формы (с точностью до замены связанных переменных).

9.1.5 Комбинаторы

Теория комбинаторов была развита российским математиком Шейфинкелем [15] перед тем как ламбда–обозначения были введены Чёрчем. Вскоре Карри переоткрыл эту теорию независимо от Шейфинкеля и Чёрча. Теорию комбинаторов так же как и λ–исчисление можно использовать для представления функций. Комбинаторы также обеспечивают хороший промежуточный код для обычных компьютеров: несколько лучших компиляторов для ленивых функциональных языков базируются на комбинаторах [8].

Существует два эквивалентных способа формализации теории комбинаторов:

1) внутри λ–исчисления;
2) как полностью независимую теорию.

Следуя первому подходу, определим комбинатор просто как λ–терм, не имеющих свободных переменных. Такой терм также называется замкнутым – он имеет фиксированное значение независимо от значения любой переменной. Оказывается любое λ–выражение равно некоторому выражению, построенному с помощью аппликации из переменных и двух комбинаторов ***K*** и ***S***. Операция β–конверсии может быть смоделирована с помощью простейших операций для ***K*** и ***S***.

Определения комбинаторов *K* и *S*

$$\boldsymbol{K} \equiv \lambda x\ y.\ x$$
$$\boldsymbol{S} \equiv \lambda f\ g\ x\ .\ (f\ x)\ (g\ x)$$

Из этих определения, используя β–конверсию получаем, что для любых термов E_1, E_2 и E_3 выполнено

$$\boldsymbol{K}\, E_1\, E_2 = E_1$$
$$\boldsymbol{S}\, E_1\, E_2\, E_3 = (E_1\, E_3)\, (E_2\, E_3)$$

Любой замкнутый λ–терм можно выразить с помощью аппликации (комбинации) двух комбинаторов ***K*** и ***S***, называемых *примитивными* комбинаторами. Определим комбинатор $\boldsymbol{I} \equiv \boldsymbol{S}\ \boldsymbol{K}\ \boldsymbol{K}$, тогда $\boldsymbol{I}\ x \equiv (\boldsymbol{S}\ \boldsymbol{K}\ \boldsymbol{K})\ x \equiv \boldsymbol{S}\ \boldsymbol{K}\ \boldsymbol{K}\ x \to (\boldsymbol{K}\ x)\ (\boldsymbol{K}\ x) \equiv \boldsymbol{K}\ x\ (\boldsymbol{K}\ x) \to$ x. В силу η–конверсии, $\boldsymbol{I} = \lambda x.\ x$.

Теорема 16 (Теорема о функциональной полноте комбинаторов). Любое λ–выражение равно λ–выражению построенному из примитивных комбинаторов ***K*** и ***S*** и переменных с помощью только операции аппликации без использовании операции λ–абстракции.

Доказательство этой теоремы см. в [1, стр. 158-171] и, в более элементарном изложении, в [10, стр. 91–97].

Например, $\lambda x.\, f\, x = \boldsymbol{S}\ (\boldsymbol{K}\, f)\ (\boldsymbol{S}\ \boldsymbol{K}\ \boldsymbol{K})$. Действительно, $(\boldsymbol{S}\ (\boldsymbol{K}\, f)\ (\boldsymbol{S}\ \boldsymbol{K}\ \boldsymbol{K}))\ y \equiv (\boldsymbol{S}\ (\boldsymbol{K}\, f)\ \boldsymbol{I})\ y \equiv \boldsymbol{S}\ (\boldsymbol{K}\, f)\ \boldsymbol{I}\ y \to ((\boldsymbol{K}\, f)\ y)\ (\boldsymbol{I}\ y) \equiv (\boldsymbol{K}\, f\, y)\ (\boldsymbol{I}\ y) \to f\ \ (\boldsymbol{I}\ y) \to f\, y$. Таким образом, искомое равенство получается в силу η–конверсии. Как произвольное λ–выражение можно транслировать в комбинаторную форму см. [8, стр. 294–296].

9.2 Ламбда–исчисление как язык программирования

В описанном виде λ–исчисление является примитивным языком. Однако его можно использовать для представления объектов и структур современного языка программирования. Идея состоит в представление этих объектов и структур в виде λ–выражений таким образом, чтобы они обладали требуемыми свойствами. Например, для представления логических значений истинности *true* и *false* и логической связки «отрицание» (*not*) λ–выражения ***true***, ***false*** и ***not*** должны удовлетворять свойствам:

$$\boldsymbol{not\ true} = \boldsymbol{false}$$
$$\boldsymbol{not\ false} = \boldsymbol{true}$$

Для того, чтобы представить конъюнкцию (*and*) соответствующее λ–выражение ***and*** должно обладать свойствами:

$$\boldsymbol{and\ true\ true} = \boldsymbol{true}$$
$$\boldsymbol{and\ true\ false} = \boldsymbol{false}$$
$$\boldsymbol{and\ false\ true} = \boldsymbol{false}$$
$$\boldsymbol{and\ false\ false} = \boldsymbol{false}$$

Подобно этому мы должны ожидать определенные свойства от λ–выражения ***or***, представляющего операцию дизъюнкции *or* для логических значений. Выбор λ–выражений для представления программных объектов, на первый взгляд, может выглядеть немотивированным, однако, все предлагаемые определения согласованы так, что они могут работать в унисон.

9.2.1 Истинностные значения и условное выражение

Определим λ–выражения ***true***, ***false***, ***not*** и условное выражение ***If*** так, чтобы выполнялись следующие свойства:

$$\boldsymbol{not\ true} = \boldsymbol{false}$$
$$\boldsymbol{not\ false} = \boldsymbol{true}$$

$$\boldsymbol{If\ true}\ E_1\ E_2 = E_1$$

$$\textbf{\textit{If false}}\ E_1\ E_2 = E_2$$

Существуют бесконечно много различных способов для представления истинностных значений и отрицания; здесь предлагаются традиционные определения.

$$\textbf{\textit{true}} \equiv \lambda x.\ \lambda y.\ x$$
$$\textbf{\textit{false}} \equiv \lambda x.\ \lambda y.\ y$$
$$\textbf{\textit{not}} \equiv \lambda t.\ t\ \textbf{\textit{false true}}$$

Легко использовать правила конверсии, чтобы показать, что данные комбинаторы обладают желаемыми свойствами. Так, например,

$\textbf{\textit{not true}} \equiv (\lambda t.\ t\ \textbf{\textit{false true}})\ \textbf{\textit{true}}$
$\rightarrow_\beta\ \textbf{\textit{true false true}} \equiv (\lambda x.\ \lambda y.\ x)\ \textbf{\textit{false true}}$
$\rightarrow_\beta\ (\lambda y.\ \ \textbf{\textit{false}})\ \textbf{\textit{true}}$
$\rightarrow_\beta\ \ \textbf{\textit{false}}$

Условное выражение можно определить следующим образом

$$\textbf{\textit{If}} \equiv \lambda x.\ \lambda y.\ \lambda z.\ x\ y\ z$$

Проверим, что данное определение обладает требуемыми свойствами:

$\textbf{\textit{If true}}\ E_1\ E_2 \equiv (\lambda x.\ \lambda y.\ \lambda z.\ x\ y\ z)\ \textbf{\textit{true}}\ E_1\ E_2 \equiv (((\lambda x.\ \lambda y.\ \lambda z.\ x\ y\ z)\ \textbf{\textit{true}})\ E_1)\ E_2 \rightarrow \textbf{\textit{true}}\ E_1\ E_2 \equiv (\lambda x.\ \lambda y.\ x)\ E_1\ E_2 \rightarrow_\beta (\lambda y.\ E_1)\ E_2 \rightarrow_\beta E_1$
$\textbf{\textit{If false}}\ E_1\ E_2 \equiv (\lambda x.\ \lambda y.\ \lambda z.\ x\ y\ z)\ \textbf{\textit{false}}\ E_1\ E_2 \equiv (((\lambda x.\ \lambda y.\ \lambda z.\ x\ y\ z)\ \textbf{\textit{false}})\ E_1)\ E_2 \rightarrow\ \ \textbf{\textit{false}}\ E_1\ E_2 \equiv (\lambda x.\ \lambda y.\ y)\ E_1\ E_2 \rightarrow_\beta (\lambda y.\ y)\ E_2 \rightarrow_\beta E_2$

Легко проверить, что конъюнкцию и дизъюнкцию можно определить так:

$$\textbf{\textit{and}} \equiv \lambda x.\ \lambda y.\ x\ y\ \ \textbf{\textit{false}}$$
$$\textbf{\textit{or}} \equiv \lambda x.\ \lambda y.\ (x\ \ \textbf{\textit{true}})\ y$$

9.2.2 Пары и кортежи

Следующие комбинаторы используются для представления пар в λ–исчислении.

$$\textbf{\textit{fst}} \equiv \lambda x.\ x\ \textbf{\textit{true}}$$
$$\textbf{\textit{snd}} \equiv \lambda x.\ x\ \textbf{\textit{false}}$$
$$(E_1, E_2) \equiv \lambda f.\ f\ \ E_1\ E_2$$

Выражение (E_1, E_2) представляет упорядоченную пару, причем доступ к первой компоненте осуществляется с помощью функции ***fst***, а вторую компоненту возвращает функция ***snd***. Проверим:

$\textbf{\textit{fst}}\ (E_1, E_2)$

$\equiv (\lambda x.\ x\ \textbf{\textit{true}})\ (\lambda f.\ f\ E_1\ E_2)$
$\to_\beta (\lambda f.\ f\ E_1\ E_2)\ \textbf{\textit{true}}$
$\to_\beta \textbf{\textit{true}}\ E_1\ E_2$
$\to_\beta (\lambda x.\ \lambda y.\ x)\ E_1\ E_2$
$\to_\beta (\lambda y.E_1)\ E_2$
$\to_\beta E_1$

Пара есть структура данных с двумя компонентами. Обобщенная структура с n компонентами называется *n–кой* или *кортежем* и легко определяется через пары.

$$(E_1, E_2, \ldots, E_n) \equiv (E_1, (E_2, (\ldots(E_{n-1}, E_n)\ldots)))$$

$(E_1, E_2, \ldots, E_n)$ есть n–кортеж с компонентами $E_1, E_2, \ldots, E_n$ и длиной n. Пары суть кортежи длиной 2. С помощью следующих выражений получаем доступ к отдельным компонентам кортежа.

$$E{\downarrow}1 \equiv \textbf{\textit{fst}}\ E$$
$$E{\downarrow}2 \equiv \textbf{\textit{fst}}\ (\textbf{\textit{snd}}\ E)$$
$$E{\downarrow}i \equiv \textbf{\textit{fst}}\ (\underbrace{\textbf{\textit{snd}}\ (\textbf{\textit{snd}}\ (\ldots(\textbf{\textit{snd}}}_{i-1 \text{ раз}}\ E)\ldots))),\ \text{если } i \text{ меньше длины } E$$
$$E{\downarrow}n \equiv \underbrace{\textbf{\textit{snd}}\ (\textbf{\textit{snd}}\ (\ldots(\textbf{\textit{snd}}}_{n-1 \text{ раз}}\ E)\ldots)),\ \text{если } n \text{ равно длине } E$$

Проверим, что эти определения правильно работают:

$(E_1, E_2, \ldots, E_n){\downarrow}1$
$\equiv \textbf{\textit{fst}}\ (E_1, (E_2, (\ldots(E_{n-1}, E_n)\ldots)))$
$\to E_1$

$(E_1, E_2, \ldots, E_n){\downarrow}2$
$\equiv \textbf{\textit{fst}}\ (\textbf{\textit{snd}}\ (E_1, (E_2, (\ldots(E_{n-1}, E_n)\ldots))))$
$\to \textbf{\textit{fst}}\ (E_2, (\ldots(E_{n-1}, E_n)\ldots)))$
$\to E_2$

В общем случае $(E_1, E_2, \ldots, E_n){\downarrow}i = E_i$ для $1 \le i \le n$.

9.2.3 Числа

Существует много способов для определения чисел в виде λ–выражений; каждый способ имеет свои преимущества и недостатки [1]. Нашей целью является определить комбинатор $\langle n \rangle$ для представления натурального числа n. Необходимо определить также примитивные арифметические операции в терминах λ–выражений. Например, нам необходимы λ–выражения ***suc, pre, add*** и ***iszero***, представляющие функции следования $n \to n+1$, предшествования $n \to n-1$, сложения и тест для нуля, соответственно. Эти λ–выражения представляют числа корректно, если они обладают следующими свойствами:

suc $\langle n \rangle = \langle n+1 \rangle$ (для всех натуральных чисел n);
pre $\langle n \rangle = \langle n-1 \rangle$ (для всех натуральных чисел $n>0$);

add <*m*> <*n*> = <*m*+*n*> (для всех натуральных чисел *m* и *n*);
iszero <0> = ***true***
iszero (***suc*** <*n*>) = ***false***

Определим натуральные числа, следуя Чёрчу. Будем использовать обозначение $f^n\ x$ чтобы представить *n*–кратное применение *f* к *x*. Например, $f^3\ x \equiv f\ (f\ (f\ x))$. Для удобства $f^0\ x$ представляет просто *x*. В общем виде,

$$E^0\ E_1 \equiv E_1$$
$$E^n\ E_1 \equiv \underbrace{(E\ (E\ (\ldots(E}_{n \text{ раз}}\ E_1)\ldots)))$$

Легко видеть, что $E^n\ (E\ E_1) = E^{n+1}\ E_1 = E\ (E^n\ E_1)$.

$$<0> \equiv \lambda f\ x.\ x$$
$$<1> \equiv \lambda f\ x.\ f\ x$$
$$<2> \equiv \lambda f\ x.\ f\ (f\ x)$$
$$\ldots$$
$$<n> \equiv \lambda f\ x.\ f^n\ x$$
$$\ldots$$

Такое представление чисел позволяет легко определить примитивные арифметические функции.

suc $\equiv \lambda n\ f\ x.\ n\ \ f\ (f\ \ x)$
add $\equiv \lambda\ m\ n\ f\ x.\ m\ f\ (\ n\ f\ x)$
iszero $\equiv \lambda n.\ n\ (\lambda\ x.$ ***false***$)$ ***true***

Проверим, что введенные комбинаторы обладают нужными свойствами:

suc <*n*> $\equiv (\lambda n\ f\ x.\ n\ \ f\ (f\ \ x))\ (\lambda f\ x.\ f^n\ x)$
$= \lambda f\ x.((\lambda f\ x.\ f^n\ x)\ \ f\ (f\ \ x))$
$= \lambda f\ x.((\lambda x.\ f^n\ x)\ \ (f\ \ x))$
$= \lambda f\ x.\ f^n\ (f\ \ x)$
$= \lambda f\ x.\ f^{n+1}\ x$
= <n+1>

iszero <0> $\equiv (\lambda n.\ n\ (\lambda x.$ ***false***$)$ ***true***$)\ (\lambda f\ x.\ x)$
$= (\lambda f\ x.\ x)\ (\lambda x.$ ***false***$)$ ***true***
$= (\lambda x.\ x)$ ***true***
= ***true***

add <*p*> <*q*> $\equiv (\lambda\ m\ n\ f\ x.\ m\ f\ (\ n\ f\ x))$ <*p*> <*q*>
$= (\lambda\ n\ f\ x.$ <*p*> $f\ (\ n\ f\ x))$ <*q*>

$= \lambda f x. <p> f (<q> f x)$
$= \lambda f x. (\lambda g y. g^p y) f (<q> f x)$
$= \lambda f x. (\lambda y. f^p y) (<q> f x)$
$= \lambda f x. (\lambda y. f^p y) ((\lambda h z. h^q z) f x)$
$= \lambda f x. (\lambda y. f^p y) ((\lambda z. f^q z) x)$
$= \lambda f x. (\lambda y. f^p y) (f^q x)$
$= \lambda f x. f^p (f^q x)$
$\equiv \lambda f x. f^{p+q} x \equiv <p+q>$

Функция предшествования ***pre*** определяется с большим трудом, чем предыдущие функции. Сам Алонсо Чёрч бился несколько месяцев над тем, чтобы определить эту функцию. Чёрч так и не справился с этой задачей и уже уверился в неполноте своего вычисления, но в 1932 г. Стефен Клини, тогда молодой аспирант, нашел решение, и это было его первым математическим результатом.

Трудность в определения функции предшествования состоит в том, что, имея терм $\lambda f x. f^n x$, надо избавиться от первой аппликации f в f^n. Чтобы достигнуть этого, предварительно определим функцию ***prefn***, оперирующую с парами и обладающую следующими свойствами:

1) ***prefn*** f (***true***, x) = (***false***, x)
2) ***prefn*** f (***false***, x) = (***false***, $f x$)

Из этого следует, что

3) (***prefn*** f)^n (***false***, x) = (***false***, $f^n x$)
4) (***prefn*** f)^n (***true***, x) = (***false***, $f^{n-1} x$) (если n>0)

Таким образом, последнее равенство, показывает, как можно получить n–1–кратную аппликацию f к x. Определим ***prefn*** следующим образом:

prefn $\equiv \lambda f p.$(***false***, (***If*** (***fst*** p) (***snd*** p) (f (***snd*** p))))

Из определения следует, что ***prefn*** f (b, x) = (***false***, (***If*** b x ($f x$)), и, следовательно:

prefn f (***true***, x)
= (***false***, (***If*** ***true*** x ($f x$)))
= (***false***, x)

prefn f (***false***, x)
= (***false***, (***If*** ***false*** x ($f x$)))
= (***false***, $f x$)

Равенство 3 докажем по индукции:
Базис индукции, n=0:

(***prefn*** f)^0 (***false***, x)
$\equiv$ (***false***, x)
$\equiv$ (***false***, $f^0 x$)

Индуктивный переход:

$(\textbf{\textit{prefn}}\ f)^{n+1}\ (\textbf{\textit{false}}, x) \equiv$
$(\textbf{\textit{prefn}}\ f)^{n}\ ((\textbf{\textit{prefn}}\ f)\ (\textbf{\textit{false}}, x))$
$= (\textbf{\textit{prefn}}\ f)^{n}\ (\textbf{\textit{false}}, f\,x)$
= (по предположению индукции) $(\textbf{\textit{false}}, f^{n}\ (f\,x))$
$\equiv (\textbf{\textit{false}}, f^{n+1}\ x)$

Равенство 4 следует из равенства 3:

$(\textbf{\textit{prefn}}\ f)^{n}\ (\textbf{\textit{true}}, x) \equiv$ (если n > 0)
$(\textbf{\textit{prefn}}\ f)^{n-1}\ ((\textbf{\textit{prefn}}\ f)\ (\textbf{\textit{true}}, x))$
$= (\textbf{\textit{prefn}}\ f)^{n-1}\ (\textbf{\textit{false}}, x)$
$= (\textbf{\textit{false}}, f^{n-1}\ x)$

Функция предшествования сейчас может быть определена так:

$$\textbf{\textit{pre}} \equiv \lambda\ n\,f\,x\ .\textbf{\textit{snd}}\ (n\ \ (\textbf{\textit{prefn}}\ \ f)\ (\textbf{\textit{true}}, x))$$

Проверим, что это правильное определение, используя экстенсиональность. Докажем, что при $n>0$ выполнено $\textbf{\textit{pre}}\ \langle n\rangle\ f\,x = \langle n-1\rangle\ f\ \ x$, откуда будет следовать, что $\textbf{\textit{pre}}\ \langle n\rangle \rightarrow_{\eta} \langle n-1\rangle$. Действительно,

$\langle n-1\rangle\ f\ \ x$
$\equiv (\lambda f\,x.\ f^{n-1}\ x)\ f\ \ x$
$= \ f^{n-1} x$

$\textbf{\textit{pre}}\ \langle n\rangle\ f\,x$
$\equiv (\lambda\ n\,f\,x\ .\textbf{\textit{snd}}\ (n\ \ (\textbf{\textit{prefn}}\ \ f)\ (\textbf{\textit{true}}, x)))\ \langle n\rangle\ f\,x$
$= \textbf{\textit{snd}}\ (\langle n\rangle\ \ (\textbf{\textit{prefn}}\ \ f)\ (\textbf{\textit{true}}, x))$
$= \textbf{\textit{snd}}\ ((\lambda f\,x.\ f^{n}\ x)\ (\textbf{\textit{prefn}}\ \ f)\ (\textbf{\textit{true}}, x))$
$= \textbf{\textit{snd}}\ (\ (\textbf{\textit{prefn}}\ \ f)^{\ n}\ (\textbf{\textit{true}}, x))$
$= \textbf{\textit{snd}}\ (\textbf{\textit{false}}, f^{n-1}\ x)$
$= f^{n-1} x$

Очевидно,

$\textbf{\textit{pre}}\ \langle 0\rangle$
$\equiv (\lambda\ n\,f\,x\ .\textbf{\textit{snd}}\ (n\ \ (\textbf{\textit{prefn}}\ \ f)\ (\textbf{\textit{true}}, x)))\ \langle 0\rangle$
$= \lambda\,f\,x\ .\textbf{\textit{snd}}\ (\langle 0\rangle\ \ (\textbf{\textit{prefn}}\ \ f)\ (\textbf{\textit{true}}, x))$ (по определению <0>)
$= \lambda\,f\,x\ .\textbf{\textit{snd}}\ ((\lambda f\,x.\ x)\ (\textbf{\textit{prefn}}\ \ f)\ (\textbf{\textit{true}}, x))$
$= \lambda\,f\,x\ .\textbf{\textit{snd}}\ (\textbf{\textit{true}}, x)$
$= \lambda\,f\,x\ .x$
$=\langle 0\rangle$

9.2.4 Рекурсивные функции

Использование рекурсивных функций – одна из важнейших особенностей функционального программирования: все итерации заменяются рекурсивными вызовами. На первый взгляд это невозможно сделать в ламбда–исчислении, поскольку в λ–исчислении

все функции анонимны, и мы не можем использовать их имена, чтобы организовать рекурсивный вызов. Как это не удивительно, но рекурсия в λ–исчислении возможна! Но этот факт, как и существование функции предшествования, был открыт только после значительных усилий.

Ключевой идеей послужило использование так называемого *комбинатора неподвижной точки*. Замкнутый λ–терм Y называется комбинатором неподвижной точки, если для любого терма f выполнено соотношение $f\ (Y\ f) = Y\ f$. Другими словами, комбинатор неподвижной точки, примененный для любой функции f, возвращает неподвижную точку этой функции. Первый такой комбинатор был найден Карри и обычно обозначается ***Y***. Этот комбинатор напоминает парадокс Рассела и поэтому называется «парадоксальным комбинатором». Если мы определим

$R = \lambda x.\ \textbf{\textit{not}}\ (x\ x)$,

то обнаружим, что

$R\ R = \textbf{\textit{not}}\ (R\ R)$

Таким образом, терм $R\ R$ служит неподвижной точкой для оператора отрицания. Для того, чтобы получить общий комбинатор неподвижной точки, мы должны заменить отрицание на произвольный терм f. Поэтому мы определяем:

$$\boldsymbol{Y} \equiv \lambda f.\ (\lambda x\ .f\ (x\ x))\ (\lambda x\ .f\ (x\ x))$$

Проверим, что ***Y*** удовлетворяет требуемым условиям. С одной стороны,

$\boldsymbol{Y}\ E \equiv (\lambda f.\ (\lambda x\ .f\ (x\ x))\ (\lambda x\ .f\ (x\ x)))\ E$
$\rightarrow_\beta\ (\lambda x\ .\ E\ (x\ x))\ (\lambda x\ .\ E\ (x\ x))$
$\rightarrow_\beta\ E\ ((\lambda x\ .\ E\ (x\ x))\ (\lambda x\ .\ E\ (x\ x)))$

С другой стороны,

$E\ (\boldsymbol{Y}\ E) \equiv E\ ((\lambda f.\ (\lambda x\ .f\ (x\ x))\ (\lambda x\ .f\ (x\ x)))\ E)$
$\rightarrow_\beta\ E\ ((\lambda x\ .\ E\ (x\ x))\ (\lambda x\ .\ E\ (x\ x)))$

Так как оба терма $\boldsymbol{Y}\ E$ и $E\ (\boldsymbol{Y}\ E)$ редуцируются к терму $E\ ((\lambda x\ .\ E\ (x\ x))\ (\lambda x\ .\ E\ (x\ x)))$, то $E\ (\boldsymbol{Y}\ E) = \boldsymbol{Y}\ E$. Таким образом, мы получили, что каждое λ–выражение E имеет неподвижную точку $\boldsymbol{Y}\ E$.

Рассмотрим теперь представление рекурсивных функций в λ–исчислении. В качестве примера, попробуем определить λ–выражение ***summa***, такое что

$$\textbf{\textit{summa}}\ \langle m\rangle = \textbf{\textit{add}}\ \langle m\rangle\ (\textbf{\textit{add}}\ \langle m–1\rangle\ (\ \ldots\ (\textbf{\textit{add}}\ \langle 1\rangle\ \langle 0\rangle)\ \ldots\ ))$$

Нетрудно увидеть, что ***summa*** должно удовлетворять уравнению:

$$\textbf{\textit{summa}}\ \langle m\rangle = \textbf{\textit{If}}\ (\textbf{\textit{iszero}}\ \langle m\rangle)\ \langle 0\rangle\ (\textbf{\textit{add}}\ \langle m\rangle\ (\textbf{\textit{summa}}\ (\textbf{\textit{pre}}\ \langle m\rangle)))$$

Пусть это имеет место, тогда, например,

$\textbf{\textit{summa}}\ \langle 2\rangle = \textbf{\textit{If}}\ (\textbf{\textit{iszero}}\ \langle 2\rangle)\ \langle 0\rangle\ (\textbf{\textit{add}}\ \langle 2\rangle\ (\textbf{\textit{summa}}\ (\textbf{\textit{pre}}\ \langle 2\rangle)))$
$= (\textbf{\textit{add}}\ \langle 2\rangle\ (\textbf{\textit{summa}}\ (\textbf{\textit{pre}}\ \langle 2\rangle)))$

= (***add*** <2> (***summa*** <1>))
= (***add*** <2> (***If*** (***iszero*** <1>) <0> (***add*** <1> (***summa*** (***pre*** <1>)))))
= (***add*** <2> (***add*** <1> (***summa*** (***pre*** <1>))))
= (***add*** <2> (***add*** <1> (***summa*** <0>)))
= (***add*** <2> (***add*** <1> (***If*** (***iszero*** <0>) <0> (***add*** <0> (***summa*** (***pre*** <0>)))))))
= (***add*** <2> (***add*** <1> <0>))
= (***add*** <2> <1>)
= <3>

Приведенное выше уравнение для ***summa*** предполагает, что терм ***summa*** может быть определен как

$$\textbf{\textit{summa}} = \lambda\, m.\ \textbf{\textit{If}}\ (\textbf{\textit{iszero}}\ m)\ \langle 0\rangle\ (\textbf{\textit{add}}\ m\ (\textbf{\textit{summa}}\ (\textbf{\textit{pre}}\ m)))$$

Но такое определение невозможно в λ–исчислении, поскольку определяемый терм не может присутствовать в правой части определения. И здесь на помощь приходит комбинатор неподвижной точки. Определим вспомогательный терм

$$\textbf{\textit{summaf}} = \lambda\, f\, m.\ \textbf{\textit{If}}\ (\textbf{\textit{iszero}}\ m)\ \langle 0\rangle\ (\textbf{\textit{add}}\ m\ (f\ (\textbf{\textit{pre}}\ m)))$$

и затем определим $\textbf{\textit{summa}} = \textbf{\textit{Y summaf}}$. Получаем искомое уравнение:

summa <*m*> = (***Y summaf***)<*m*>
= ***summaf*** (***Y summaf***) <*m*>
= (по определению ***summa***) ***summaf summa*** <*m*>
= (λ *f m*. ***If*** (***iszero*** *m*) <0> (***add*** *m* (*f* (***pre*** *m*)))) ***summa*** <*m*>
= ***If*** (***iszero*** <*m*>) <0> (***add*** <*m*> (***summa*** (***pre*** <*m*>)))

Уравнение вида $f\ x_1 \ldots x_n = E$ называется рекурсивным, если f имеет свободное вхождение в E. Комбинатор ***Y*** обеспечивает общий путь решения такого уравнения. Начинаем с уравнения в форме $f\ x_1 \ldots x_n = \sim f \sim$, где $\sim f \sim$ есть некоторое λ–выражение, содержащее f. Затем определяем f как

$$f = \textbf{\textit{Y}}\ (\lambda\, f\, x_1 \ldots x_n\ .\ \sim f \sim)$$

Проверим, что f удовлетворяет необходимому уравнению:

$f\, x_1 \ldots x_n$
= (***Y*** (λ $f\, x_1 \ldots x_n$. $\sim f \sim$)) $x_1 \ldots x_n$
= (по свойству ***Y***) (λ $f\, x_1 \ldots x_n$. $\sim f \sim$) (***Y*** (λ $f\, x_1 \ldots x_n$. $\sim f \sim$)) $x_1 \ldots x_n$
= (по определению *f*) (λ $f\, x_1 \ldots x_n$. $\sim f \sim$) $f\, x_1 \ldots x_n$
= $\sim f \sim$

Хотя с математической точки зрения использование комбинатора Карри совершенно безукоризненно, но с точки зрения программирования вычислительный переход от выражения E (***Y*** E) к выражению ***Y*** E и обратно нельзя осуществить только с помощью β–конверсии. По этой причине более удобен комбинатор неподвижной точки, предложенный Тьюрингом:

$$\textbf{\textit{T}} \equiv (\lambda x\, y.\ y(x\ x\ y))\ \ (\lambda x\, y.\ y(x\ x\ y))$$

Для того, чтобы проверить необходимое свойство, обозначим терм $\lambda x\, y.\, y(x\, x\, y)$ через ***a***. Тогда имеем

$$\boldsymbol{T}\, f \equiv (\lambda x\, y.\, y(x\, x\, y))\ \boldsymbol{a}\, f \to_\beta f\ (\boldsymbol{a}\ \boldsymbol{a}\ f) \equiv f\ (\boldsymbol{T}\, f)$$

9.2.5 Функции с несколькими аргументами

В математических обозначениях применение n–местной функции f к аргументам $x_1 \ldots x_n$ записывается как $f\ (x_1, \ldots, x_n)$. Существует два способа представления n–кратной аппликации в λ–исчислении:

1) как $(f\, x_1 \ldots x_n)$, или
2) как аппликация f к n–ке $(x_1, \ldots, x_n)$.

В первом случае f применяется к своим аргументам по–очереди. Русский математик Шейфинкель заметил, что не обязательно вводить функции более чем одной переменной [15]. Действительно, для функции, скажем от двух переменных, $f(x,y)$ мы можем рассмотреть функцию g_x с соотношением $g_x\ (y) = f(x,y)$, а затем f' с соотношением $f'(x) = g_x$. Отсюда $(f'(x))(y) = f(x,y)$. Позднее Карри [16] переоткрыл это свойство и поэтому сейчас сведение функций с несколькими переменными только к функциям одного переменного носит название *карринг*. Функции ***and***, ***or*** или ***add***, определенные ранее, применяют каррринг. Преимущество функций с каррингом заключается в том, что такие функции можно применять частично с меньшим числом аргументом, чем требуется по определению этих функций. Например, ***add*** <1> есть результат частичного применения функции ***add*** к <1> и обозначает функцию $n \to n$+1.

Хотя часто удобно представлять n–местные функции с помощью карринга, полезно также иметь возможность представлять аргументы с помощью единственной n-ки. Например, вместо представления сложения λ–термом ***add***, так что выполняется условие

$$\boldsymbol{add}\ \langle m\rangle\ \langle n\rangle = \langle m+n\rangle$$

может быть более удобно представить λ–термом ***sum***, для которого выполнено

$$\boldsymbol{sum}\ (\langle m\rangle, \langle n\rangle) = \langle m+n\rangle$$

Таким образом, n–местные функции можно определить как с каррингом, так и без карринга.

Можно определить два комбинатора, применяя которые к функциям, можно менять наличие карринга у последних.

$$\boldsymbol{curry} \equiv \lambda\, f\, x\, y.\, f\ (x,\, y)$$
$$\boldsymbol{uncurry} \equiv \lambda\, f\, p.\, f\ (\boldsymbol{fst}\ p)\ (\boldsymbol{snd}\ p)$$

Имеем для произвольного выражения E:

$$\boldsymbol{curry}\ (\boldsymbol{uncurry}\ E) = E$$
$$\boldsymbol{uncurry}\ (\boldsymbol{curry}\ E) = E$$

Проверим первое равенство:

$$\boldsymbol{curry}\ (\boldsymbol{uncurry}\ E)$$
$$\equiv (\lambda\, f\, x\, y.\, f\ (x,\, y))\ (\boldsymbol{uncurry}\ E)$$

$= \lambda\ x\ y.((\textbf{\textit{uncurry}}\ E)\ (x, y))$
$\equiv \lambda\ x\ y.\ ((\lambda f\ p.\ f\ (\textbf{\textit{fst}}\ p)\ (\textbf{\textit{snd}}\ p))\ E\ (x, y))$
$= \lambda\ x\ y.\ (E\ (\textbf{\textit{fst}}\ (x, y))\ (\textbf{\textit{snd}}\ (x, y)))$
$= \lambda\ x\ y.\ E\ x\ y$
$\rightarrow_\eta E$

Аналогично доказывается второе равенство. Нетрудно увидеть, что

$$\textbf{\textit{sum}} = \textbf{\textit{uncurry add}}$$
$$\textbf{\textit{add}} = \textbf{\textit{curry sum}}$$

9.2.6 Представление вычислимых функций

Один из подходов к формализации понятия алгоритма принадлежит Гёделю и Клини (1936). Основная идея Гёделя состояла в том, чтобы получить все вычислимые функции из существенно ограниченного множества базисных функций с помощью простейших алгоритмических средств.

Пусть **N** обозначает множество натуральных чисел {0,1,2,...}. Объекты, которые мы будем рассматривать, будут функциями с областью определения $D_f \subseteq \mathbf{N}^k$ (k - целое положительное число) и с областью значений $R_f \subseteq \mathbf{N}$. Такие функции будем называть *k-местными частичными*. Слово «частичная» должно напомнить о том, что функция определена на подмножестве $\mathbf{N}^k$ (конечно, в частном случае может быть $D_f = \mathbf{N}^k$, тогда функция называется *всюду определенной*).

Множество исходных функций таково ($\mathbf{x} \in \mathbf{N}^k$):

- постоянная функция $0(\mathbf{x}) = 0$;
- одноместная функция следования $s(x) = x+1$;
- функции проекций pr_i, $1 \le i \le k$, $pr_i(\mathbf{x}) = x_i$.

Нетривиальные вычислительные функции можно получать с помощью композиции (суперпозиции) уже имеющихся функций. Этот способ явно алгоритмический.

- Оператор суперпозиции. Говорят, что k-местная функция $f(x_1, x_2, ..., x_k)$ получена с помощью суперпозиции из m-местной функции $\varphi(y_1, y_2, ..., y_m)$ и k-местных функций $g_1(x_1, x_2, ..., x_k)$, $g_2(x_1, x_2, ..., x_k)$, ..., $g_m(x_1, x_2, ..., x_k)$, если
  $f(x_1, x_2, ..., x_k) = \varphi(g_1(x_1, x_2, ..., x_k), g_2(x_1, x_2, ..., x_k), ..., g_m(x_1, x_2, ..., x_k))$.

Второй (несколько более сложный) способ действует так.

- Примитивная рекурсия. При $n \ge 0$ из n-местной функции f и $(n+2)$-местной функции g строится $(n+1)$-местная функция h по следующей схеме:
  $h(x_1, ..., x_n, 0) = f(x_1, ..., x_n)$,
  $h(x_1, ..., x_n, y+1) = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y))$.

При $n=0$ получаем (a - константа)
$h(0) = a$;
$h(y+1) = g(y, h(y))$.

Две упомянутых способа позволяют задать только всюду определенные функции. Частично-определенные функции порождаются с помощью третьего гёделева механизма.

- Оператор *минимизации*. Эта операция ставит в соответствие частичной функции f: $\mathbf{N}^{k+1} \rightarrow \mathbf{N}$ частичную функцию h: $\mathbf{N}^k \rightarrow \mathbf{N}$, которая определяется так:
  1) область определения $D_h = \{<x_1,..., x_k> \mid$ существует $x_{k+1} \ge 0$, $f(x_1,..., x_k, x_{k+1}) = 0$ и $< x_1,..., x_k, y> \in D_f$ для всех $y \le x_{k+1}\}$;
  2) $h(x_1,..., x_k)$ = наименьшее значение y, при котором $f(x_1,...,x_k, y) = 0$.

Оператор минимизации обозначается так $h(x_1,..., x_k) = \mu y[f(x_1,..., x_k, y) = 0]$. Очевидно, что даже если f всюду определено, но нигде не обращается в 0, то $\mu y[f(x_1,..., x_k, y) = 0]$ нигде не определено. Естественный путь вычисления $h(x_1,..., x_k)$ состоит в подсчете значения

$f(x_1,..., x_k, y)$ последовательно для $y = 0, 1, 2,...$ до тех пор, пока не найдется y, обращающее $f(x_1,..., x_k, y)$ в 0. Этот алгоритм не остановится, если $f(x_1,..., x_k, y)$ нигде не обращается в 0.

Все функции, которые можно получить из базисных функций за конечное число шагов только с помощью трех указанных механизмов, называются *частично-рекурсивными*. Если функция получается всюду определенная, то тогда она называется *общерекурсивной*. Если функция получена без механизма минимизации, то в этом случае она называется *примитивно-рекурсивной*.

После того как Чёрч ввел λ–исчисление, Клини доказал, что любую частично–рекурсивную функцию можно представить в виде λ–терма.

Говорят, что частичная функция φ с k аргументами *λ–определима* термом M, когда

- $M<n_1><n_2>...<n_k> = <\varphi(n_1,n_2,...,n_k)>$, если значение $\varphi(n_1,n_2,...,n_k)$ определено, и
- $M<n_1><n_2>...<n_k>$ не имеет нормальной формы, если $\varphi(n_1,n_2,...,n_k)$ не определено.

Теорема 17 (Теорема Клини) [1, с. 189]. Частичная функция является частично–рекурсивной тогда и только тогда, когда она λ–определима.

Для доказательства этой теоремы необходимо, в частности, представление функции предшествования $n \to n–1$ в виде комбинатора ***pre***.

Таким образом, мы получаем, что множество λ–определимых функций также является формализацией интуитивного понятия алгоритма. Это утверждение является содержанием известного *тезиса Черча* (в узком смысле) [5].

9.2.7 Расширение ламбда–исчисления

Хотя возможно представлять структуры данных в виде λ–выражений, но этот способ является неэффективным. Например, большинство компьютеров аппаратно поддерживают арифметику и разумно использовать такие возможности вместо β–конверсии при работе с числами. Математически строгий способ взаимодействия исчисления с компьютерными вычислениями осуществляется с помощью так называемых δ–*правил*. Идея состоит в том, чтобы добавить некоторое множество новых констант и определить правила, называемые δ–правилами, для редуцирования аппликаций, содержащих эти константы. Например, можно добавить натуральные числа и новую константу «+» вместе с δ–правилом:

$$+ \; m \; n \to_\delta m+n$$

При добавлении новых констант и правил надо быть уверенным, что остается справедливой теорема Чёрча–Россера [1, стр. 84].

9.2.8 Типовое ламбда–исчисление

Типовое ламбда–исчисление необходимо потому, что, во-первых, функция не является полностью определенной, пока не указаны ее область определения (множество всех допустимых входных данных) и область значений (множество всех возможных выходов) и, во-вторых, структура терма, представляющего функцию, должна нести информацию о ее области определения и области значений.

Чтобы реализовать это на практике, надо определить некоторые выражения, которые называются типами и предназначены для обозначения множеств. Сначала выберем некоторые символы для обозначения элементарных типов. Например, пусть *Integer* обозначает множество всех целых чисел, а *Bool* обозначает множество истинностных значений. Из любых двух типов a и b можно построить новый тип

$$a \to b,$$

обозначающий множество функций, которые определены на a и принимают значения из b. При обозначении сложных функциональных типов скобки опускаются таким образом, что, например,

$$a \to (b \to c) \equiv a \to b \to c.$$

Вариант ламбда–исчисления, в котором каждый терм помечен некоторым типом, называется *типовым ламбда–исчислением*. Функциональная аппликация $f\ g$ в типовом ламбда–исчислении синтаксически правильна, если терм f имеет тип $s \to t$, где тип терма g есть s (или экземпляр s в полиморфном языке) и t – любой тип. Если множество типов ограничено так, что терм не может применяться сам к себе, то в языке исчезает один из видов незавершающихся вычислений. Большинство функциональных языков, например, Haskell, использует типовое ламбда–исчисление как промежуточный код при трансляции на машинный язык.

10 Ленивое функциональное программирование

10.1 Haskell

Традиционно при изучении функционального программирования пользуются языком программирования Лисп. Для не новичков в программировании язык Лисп вызывает определенный дискомфорт, чему способствует и его синтаксис с множеством скобок. Те, кто уже программировал на привычных процедурных языках (скажем, на Паскале) стараются в первую очередь воспользоваться в Лиспе уже привычными конструкциями (присваиваниями, циклами) – благо, что различные диалекты Лиспа, как правило, этим располагают. После изучения курса «Функциональное программирование» у студентов остается впечатление только о языке – «вычурный язык, лежащий на обочине главного направления в программировании, применяемый главным образом в области искусственного интеллекта, поскольку там сложные символьные структуры данных».

Но такое впечатление не имеет никакого отношения к функциональному программированию! Парадигма функционального программирования – это очень мощный и красивый подход к программированию, который позволяет очень быстро создавать эффективные и надежные, компактные и легко модифицируемые программы самого широкого профиля приложений. Функциональное программирование постоянно находится в интенсивном развитии. В последнее время реализованы очень мощные и удобные функциональные языки, обладающие новыми возможностями по сравнению с Лиспом. Эти языки не провоцируют программиста воспользоваться привычными методами – такие вещи как операторы присваивания и циклы там просто отсутствуют. Поэтому программисту приходится действительно следовать принципам функционального программирования.

Первый пик интереса к функциональным языкам ним приходится на конец 70-х – начало 80-х годов 20 века. В середине 80 годов не существовало «стандарта» на чистый ленивый функциональный язык программирования. В 1987 г. был создан комитет по разработке такого стандарта и результатом его работы явился язык Haskell. Появление языка Haskell вновь вызвало интерес к функциональным языкам.

Haskell – общего назначения, чисто функциональный язык программирования, включивший в себя много недавних инноваций в проектировании языков программирования. Haskell имеет функции высокого порядка, нестрогую (ленивую) семантику, статическое полиморфное типизирование, определяемые пользователем алгебраические типы данных, сопоставление с образцом, порождение списков с помощью форм (как в теории множеств), модульную систему, систему ввода–вывода с помощью монад и богатое множество встроенных типов данных, включающих бесконечнозначные целые числа. Язык является кульминацией и консолидацией многих лет исследований в области ленивых функциональных языков.

Создание больших программных систем очень дорого и трудно. Еще более трудоемко и дорогостояще сопровождение таких систем. Для преодоления кризиса в программировании возникла методология объектно-ориентированного программирования. Важно отметить, что функциональное программирование, в частности, язык Лисп, одно из первых реализовало существенные свойства методологии объектно-ориентированного программирования: инкапсуляцию, наследование и полиморфизм.

Функциональные языки, такие как Haskell позволяют разрабатывать и сопровождать программные системы быстро и дешево. Язык Haskell имеет широкий спектр приложений. В частности, он удобен, когда требуется высокая модифицируемость программ.

Web-сайт языка Haskell (www.haskell.org) дает много полезных ресурсов, включающих:

- Определение языка Haskell и библиотек.
- Обучающий материал по программированию на языке Haskell.
- Свободно распространяемые реализации языка.

- Приложения, написанные на языке Haskell.

Познакомимся с основными чертами языка Haskell.

В функциональном программирование одно из основных понятий это понятие величины. *Величина* – абстрактная сущность (объект), что мы рассматриваем как ответ. Величина может быть атомарной или составной (сложной). Атомарные величины, обычные для функциональных языков, суть различные виды чисел, символы (например, `'a'`) и логические значения `True` и `False`. Составными структурами в языке Haskell являются список – конечная последовательность величин одной природы (одного типа) – примером может служить список из трех чисел `[1,2,3]`, пара (например, `('a',5)` – компонентами пары в данном случае служат символ и целое число), функция (математическая функция $x \to x+1$ записывается на языке как `\x -> x+1` – так называемое «λ-обозначение») и другие.

Все величины в языке Haskell являются так называемыми объектами первого класса – они могут быть фактическими аргументами функций, возвращаться как результат вычисления функций, размещаться в структурах данных и т. п.

Неформально, *выражение* – символ или группа символов, представляющих некоторый математический процесс или какое-то количество. В чисто функциональных языках все вычисления выполняются через оценку выражений, чтобы выдать какие-то величины.

Как уже говорилась ранее, альтернативой операторам присваивания и циклам в функциональном языке является рекурсия. Например, вычисление наибольшего общего делителя двух неотрицательных целых чисел может быть записано математически в следующей рекурсивной форме:

$$gcd\,(u, v) = \begin{cases} u, \text{ если } v=0, \\ gcd\,(v, u \bmod v), \text{ если } v \neq 0, \\ \text{и не определено при } v=0,\ u=0. \end{cases}$$

Но в языке Haskell запись почти похожа:

```
gcd 0 0 = error "gcd 0 0 is undefined"
gcd u 0 = u
gcd u v = gcd v (u `rem`v)
```

Функциями первого порядка называются функции, аргументами для которых служат величины, сами не являющиеся функциями. Функции второго порядка оперируют с функциями первого порядка и т. д. *Функцией высокого порядка* называется функция, не являющаяся функцией первого порядка, другими словами, такая функция имеет в качестве аргумента какую-нибудь функцию.

Мы суммируем особенности функциональных языков программирования и функциональных программ следующим образом:

- Все программы и процедуры являются функциями и ясно различают входящие величины (параметры) от своих выходящих величин (результатов).
- Нет переменных или присваиваний – переменные заменяются параметрами.
- Нет циклов – циклы заменяются рекурсивными вызовами.
- Величина функции зависит только от величины ее параметров и не зависит от порядка вычисления или пути выполнения, который привел к вызову функции (прозрачность по ссылкам).
- Функции есть одна из структур данных.

Хотя возможно использовать императивные языки, чтобы написать некоторые функциональные программы, присущие им ограничения этих языков делает трудным или

невозможным, чтобы преобразовать все программы в функциональный стиль. В общем (и традиционно) императивные языки имеют следующие ограничения:

- структурные величины, такие как массивы и записи, не могут быть результатами вызовов функций;
- нет способа построить величину структурного типа прямо;
- функции не являются данными, так что нет функций высокого порядка.

Каждая величина в функциональном языке имеет связанный (ассоциированный) *тип*. Интуитивно, мы можем рассматривать типы как множества допустимых значений, которые может иметь величина. Вообще говоря, типы могут явно или неявно присутствовать в языке. Точно так же, как выражения обозначают величины, типовые выражения – синтаксические термы, которые обозначают типовые величины (или просто *типы*). Примеры выражений типа включают атомарные типы `Integer` (целые бесконечной точности), `Char` (символы), `Integer->Integer` (функции с областью определения и областью значений `Integer`), а также структурные типы `[Integer]` (однородные списки целых чисел) и `(Char,Integer)` (пары символ, целое число).

Типы в некотором смысле описывают величины и ассоциация типов с величинами называется типизированием. Используя примеры типов, приведенных ранее, мы можем записать типизирование следующим образом:

```
5  :: Integer
'a' :: Char
inc :: Integer -> Integer
[1,2,3] :: [Integer]
('b',4) :: (Char,Integer)
```

Лексема «`::`» может быть прочитана как «имеет тип».

Функции в Haskell нормально определяются последовательностью уравнений. Например, функция `inc` может определяться одним уравнением (первая строка задает определение типа функции – она может отсутствовать):

```
inc :: Integer -> Integer
inc n          = n+1
```

Вышеприведенное определение функции `gcd` содержит три уравнения. Суперпозиция функций, которая в математике записывается как $f\,(g\,(x))$, на языке Haskell представляется в виде выражения $f\ g\ x$. Вызов математической функции с несколькими аргументами, например $f(x,y+x,5)$ в языке выглядит как $f\ x\ (y+x)\ 5$.

Как пример определенной пользователем функции, что действует на списках, рассмотрим задачу подсчета количества элементов в списке:

```
length                  :: [a] -> Integer
length []               =  0
length (x:xs)           =  1 + length xs
```

Некоторые разъяснения: конструктор «:» позволяет создавать списки, добавляя элемент в начало списка (в данном случае x – первый элемент списка, xs – остальные элементы); [] обозначает пустой список. Определение типа функции `length` говорит, что функция является полиморфной: список содержит элементы любого типа `a`.

Мы можем прочитать уравнения словами: «длина пустого списка равна нулю, а длина списка, чей первый элемент – `x` и хвост (остальные элементы) есть `xs`, равна 1 плюс длина `xs`».

Функциональное применение (аппликация) имеет наивысший приоритет среди операций, поэтому `1 + length xs` понимается синтаксически как `1 + (length xs)`. В

общем случае, Вы должны неатомарные аргументы функции помещать в круглые скобки. Это справедливо и для аргументов, которые сами являются функциональной аппликацией. Например, `f g x` - функция `f` имеет аргументы `g` и `x`; на других языках это пишется как `f(g,x)`. Вместе с тем, `f(g x)` представляет функцию `f` с аргументом `(g x)`, или, в обычной записи, `f(g(x))`.

Хотя интуитивно, но пример с функцией `length` выделяет важный аспект языка Haskell, который носит название «*сопоставление с образцом*». Левые стороны уравнений содержат образцы, в данном случае, `[]` и `x:xs`. В вызовах функций эти образцы должны соответствовать фактическим параметрам на довольно интуитивном уровне (`[]` соответствует только пустому списку, и `x:xs` может быть успешно соответствовать любому списку, по крайней мере, с одним элементом, причем `x` – обозначение для первого элемента и `xs` – хвост списка). Если сопоставление получается, то правая сторона уравнения оценивается и возвращается как результат вызова. Если сопоставление терпит неудачу, то используется следующее уравнение, и если все уравнения терпят неудачу, то возникает ошибка.

Поскольку Haskell – функциональный язык, нужно ожидать, что функции играют основную роль, и на самом деле это так. Мы уже познакомились с некоторыми способами определения функций. Перечислим все способы.

Функции могут быть определены на верхнем уровне программы, как, например:

```
f x = x*x + 3         -- используя явно аргумент

f = \x -> x * x + 3   -- используя ламбда-абстракцию
```

Заметим, символ равенства обозначает равенство по определению (это не является ни диструктивным присваиванием, ни предикатом равенства).

Определение функции может также иметь форму сопоставления с шаблоном (образцом), или использовать несколько уравнений или логические предохранители.

Функции также могут быть локально определены внутри выражений, используя `let`–конструкцию, например:

```
f x y z = let sq i = i * i
          in sq x * sq y * sq z
```

Локальные определения внутри другого определения также возможны через `where`–конструкцию, например:

```
f x y z = sq x * sq y * sq z where
                              sq v = v * v
```

Могут быть введены анонимные функции, используя ламбда–выражение

```
\x y z -> x * y * z
```

которое эквивалентно следующему выражению

```
let  h x y z = x*y*z in h
```

Изучим некоторые аспекты функций в языке Haskell. Сначала, рассмотрим определение функции, которая суммирует свои два аргумента:

```
add                    :: Integer -> Integer -> Integer
add x y                =  x + y
```

Это пример функции с каррингом. Для того, чтобы получить тот же результат без карринга, мы могли бы использовать кортеж:

```
add1                    :: (Integer,Integer) -> Integer
add1 (x,y)              = x + y
```

Нетрудно увидеть, что эта версия сложения есть в действительности просто функция одного аргумента!

Применение `add` имеет форму `add` e_1 e_2, и это эквивалентно `(add` e_1`)` e_2, так как функциональная аппликация (применение) является лево–ассоциативной. Другими словами, применение `add` к одному аргументу возвращает новую функцию, которая затем применяется ко второму аргументу. Это соответствует типу `add`, который есть

`Integer->Integer->Integer`,

что эквивалентно

`Integer->(Integer->Integer)`;

то есть конструктор типа `->` является право–ассоциативным.

10.2 Функции высших порядков

Используя `add`, мы можем определить функцию `inc` способом отличным от предыдущего:

```
inc :: Int -> Int
inc = add 1

> inc 3  -- это вызов функции inc
4        -- а это результат вызова
```

Мы видим пример частичной аппликации функции с каррингом и, кроме того, получаем, что в результате вычисления может возвращаться функция. Давайте рассмотрим случай, в котором полезно использовать функцию как аргумент. Функция `map` – отличный пример:

```
map                     :: (a->b) -> [a] -> [b]
map f  []               =  []
map f (x:xs)            =  f x : map f xs
```

Функция `map` из списка создает новый список, заменяя элементы на результат применения к ним функции `f`. Функциональная аппликация имеет наивысший приоритет среди всех инфиксных операторов и, таким образом, правая сторона второго уравнения грамматически разбирается как `(f x) : (map f xs)`. Функция `map` – полиморфная и ее тип несомненно указывает, что первый аргумент – функция.

```
e1 :: [Int]
e1 = map (add 1) [1,2,3]

> e1
[2,3,4]
```

Следующее определение эквивалентно `e1`

```
e2 :: [Int]
e2 = map inc [1,2,3]

> e2
[2,3,4]
```

Используя карринг, функция от n аргументов может быть рассмотрена как функция от одного аргумента, которая возвращает функцию от n-1 аргументов.

Функция подобная

```
f x y z = sq x * sq y * sq z where
                              sq v = v * v
```

что требует три целых аргумента и возвращает целый результат, имеет тип

```
Int -> Int -> Int -> Int
```

Однако это не означает, что такая функция должна всегда появляться точно с тремя аргументами в выражении. Вместо этого функция может появиться с любым меньшим количеством аргументов и в этом случае она рассматривается как анонимная функция (функция без имени), которую в свою очередь можно применить к недостающим аргументам. Это означает, что `(f 7)` есть правильное выражение типа `Int -> Int -> Int`, и `(f 7 13)` обозначает функцию типа `Int -> Int`. Заметим, что синтаксический разбор выражения подобного `f 7 13 0` дает `((f 7) 13) 0`. И поэтому выражение для типа `Int -> Int -> Int -> Int` анализируется как `Int -> (Int -> (Int -> Int))`.

Использование функций высших порядков позволяет достигать высокой абстракции и создавать «универсальные» функции. Одной из такой функции является `foldr`.

Рассмотрим задачи со списками целых чисел. Определим рекурсивную функцию `sum`, возвращающую сумму элементов списка. Функция должна быть определена для двух видов аргумента: пустого списка и списка, содержащего элементы. Так как сумма пустого множества чисел, очевидно, ноль, то мы определяем:

```
sum [] = 0
```

В общем случае чтобы вычислить сумму элементов списка `x : xs`, мы вычисляем сумму хвоста списка – подсписка, начинающего со второго элемента, – и прибавляем значение первого элемента:

```
sum (x : xs) = x + sum xs
```

Изучая это определение, мы обнаруживаем, что только его части, ограниченные рамками, связаны именно с вычислением суммы.

```
          +---+
sum [] = | 0 |
          +---+
                   +---+
sum (x : xs) = x | + | sum xs
                   +---+
```

Это означает, что вычисление суммы может быть модуляризовано с помощью «склеивания» вместе общего рекурсивного образца и частей, ограниченных прямоугольниками. Этот рекурсивный образец называется `foldr` и функция `sum` может быть записана в виде

```
sum = foldr (+) 0
```

где операторное обозначение суммы заменено функциональным обозначением:

```
(+) x y = x + y
```

Определение функции `foldr` может быть выведено с помощью параметризации определения функции `sum`:

```
(foldr f x) [] = x
(foldr f x) (h : t) = f  h  ((foldr f x) t)
```

Мы пишем скобки `(foldr f x)`, чтобы было понятнее, на что заменено выражение `sum`. Удобно скобки опускать и поэтому `((foldr f x) t)` может быть переписано как `(foldr f x t)`. Отметим следующий факт: функция от трех аргументов, такая как `foldr`, может использоваться только с двумя аргументами, возвращая в качестве результата функцию от одного остающегося аргумента. Так вызов `foldr (+) 0` возвращает функцию `sum` от одного аргумента – списка.

Проведенная модуляризация функции `sum` подсказывает использовать ее части в других ситуациях. Наиболее интересная часть есть `foldr`, которую мы теперь можем использовать для нахождения произведения всех элементов списка без программирования:

```
product = foldr (*) 1
```

Точно таким же образом, мы можем написать логические функции для списка булевских значений

```
and, or   :: [Bool] -> Bool
and        = foldr (&&) True
or         = foldr (||) False
```

Один из способов понимать `foldr f x` – считать это функцией, которая заменяет все символы «`:`» в списке символом `f`, и все вхождения пустого списка `[]` символом `x`. Если взять список `[1,2,3]` в качестве примера, то его можно переписать в виде

```
(:) 1 ((:) 2 ((:) 3 []))
```

и функция (`foldr (+) 0`) конвертирует это выражение в выражение

```
(+) 1 ((+) 2 ((+) 3 0))
```

и (`foldr (*) 1`) преобразует это выражение в соответствующее выражение

```
(*) 1 ((*) 2 ((*) 3 1))
```

Очевидно, что (`foldr (:) []`) просто копирует список. Конкатенацию (соединение) двух списков мы можем определить как

```
append a b = foldr (:) b a
```

Пример:

```
append [1,2] [3,4]
⇒ foldr (:) [3,4] [1,2]
⇒ (foldr (:) [3,4]) ((:) 1 ((:) 2 []))
```

(заменили `(:)` на `(:)` и `[]` на `[3,4]`)

```
⇒ (:) 1 ((:) 2 [3,4])
⇒ [1,2,3,4]
```

Декомпозируя простую функцию (`sum`) как комбинацию «функции высокого порядка» и некоторого простого аргумента, мы пришли к такой функции (`foldr`), которая может быть использована для определения многих функций для списков без больших

усилий программирования. Все стандартные функции со списками можно реализовать с помощью функции `foldr`. Нетрудно определить тип функции в общем случае:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Приведем ряд примеров использования функции `foldr`. Функцию, которая удваивает все элементы списка, можно определить так:

```
doubleall = foldr ((:) . double) []
      where double n = 2*n
```

Функция `map` применяет функцию `f` ко всем элементам списка. Эта функция является другим примером общей и полезнейшей функции в функциональном программировании.

```
map f     = foldr ((:) . f) []
```

Обращение списка (переписывание элементов списка в противоположном порядке):

```
reverse = foldr postfix []
          where postfix a x = x ++ [a]
```

Функция `filter` выбирает из списка только те элементы, которые удовлетворяют данному предикату:

```
filter  :: (a -> Bool) -> [a] -> [a]
filter p = foldr f []
           where f x xs | p x       = x:xs
                        | otherwise = xs
```

Количество элементов в списке:

```
length :: [a] -> Int
length = foldr  (\x, n -> 1+n)  0
```

Все эти возможности могут быть получены, потому что индивидуальные функции в привычных императивных языках могут быть представлены в функциональном языке как комбинация общей функции высокого порядка и некоторой частной специализированной функции. Действия, которые мы проделывали, чтобы выделить функцию высокого порядка называются *абстракция вычислений*. Мы обобщаем функции**,** имеющих одинаковый вид, т. е. определения этих функций с точки зрения абстракции вычислений сходны по строению, но которые довольно различны с точки зрения производимых действий.

Однажды определенная такая функция высокого порядка позволяет очень легко запрограммировать многие различные действия. Как только вводится новый тип данных полезно сразу определить соответствующую общую функцию (подобную `foldr`) для манипулирования с новым типом данных. Тем самым обрабатывать данные становится достаточно просто, и также локализуется знание о представлении данных нового типа.

Кроме того, функции высокого порядка позволяют избавиться от явной рекурсии.

Пример полезности функций высокого порядка. В 1979 году программист Richard Waters изучал Fortran Scientific Subroutine Library, которая представляет собой важнейший пакет, состоящий из тысяч строк кода, и широко использующий в то время. Он обнаружил, что если Fortran имел бы в своем составе три основные функции высокого порядка: `map`, `filter` и `foldr`, то около 60% кода можно было переписать как вызовы этих функций. В этом случае значительно уменьшился бы объем библиотеки, и это также означает, что повышение быстродействия пакета во многом зависело бы от оптимизации данных функций.

10.3 Ленивые вычисления

Строгая семантика обычных языков программирования требует, чтобы перед вычислением значения функции, значения аргументов функции были вычислены. Нестрогая семантика подразумевает, что аргументы функции будут вычисляться только тогда, когда возникнет абсолютная необходимость в их значениях. Такие функциональные языки называются «ленивыми». Реализация ленивых функциональных языков – это достижение последнего времени. Язык Haskell является ленивым языком. Рассмотрение следующего примера пояснит нам суть ленивых вычислений. Пусть вызов функции

```
g 55
```

есть незавершаемое (бесконечное) или ошибочное вычисление (скажем содержащее деление на ноль), но вычисление

```
f  7 (g 55) 0
```

может быть, тем не менее, успешным в ленивом языке, если функция `f` такова, что игнорирует свой второй аргумент (если первый, скажем, равен 7). Это свойство гибкости языка может быть очень полезно также в оперировании с бесконечными структурами данных.

Важным следствием ленивых вычислений появляется принципиальная возможность создавать бесконечные структуры данных.

```
ones :: [Int]
ones = 1 : ones

numsFrom :: Int -> [Int]
numsFrom n = n : numsFrom (n+1)
```

Таким образом, `numsFrom n` – бесконечный список последовательных целых, начинаемый с `n`. Используя этот список, мы можем создать бесконечный список квадратов:

```
squares :: [Integer]
squares                  = map (^2) (numsfrom 0)
```

Когда мы собираемся напечатать что-нибудь, нам необходима функция, которая отрезает от бесконечных списков конечную часть подходящего размера. Функция take возвращает первые k элементов списка:

```
take :: Int -> [a] -> [a]
take 0 x      = []                  -- два базовых случая: k = 0
take k []     = []                  -- или список пустой
take k (x:xs) = x : take (k-1) xs
```

Сейчас можно напечатать часть списка:

```
> take 5 ones
[1,1,1,1,1]

> take 5 (numsFrom 10)
[10,11,12,13,14]

> take 5 squares
[0,1,4,9,16]
```

Функция `zip` превращает два списка в список упорядоченных пар. Если списки имеют различную длину, то длина результата совпадает с длиной более короткого списка.

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

```
zip xs ys = []   -- один из списков есть []

> zip [1,2,3] [4,5,6]
[(1,4),(2,5),(3,6)]
> zip [1,2,3] ones
[(1,1),(2,1),(3,1)]
```

Определение `ones` выше является примером определения циклического списка. В большинстве обстоятельств ленивые вычисления имеют важное значение для эффективности, так как при реализации можно хранить бесконечные списки как циклические структуры; таким способом сохраняется пространство.

В качестве другого примера использования цикличности, можно взять последовательность Фибоначчи. Числа Фибоначчи можно эффективно вычислить как следующую бесконечную последовательность с помощью формы[2]:

```
fib :: [Integer]
fib = 1 : 1 : [x+y | (x,y) <- zip fib (tail fib)]
```

Первые 10 чисел Фибоначчи получаются просто:

```
> take 10 fib
[1,1,2,3,5,8,13,21,34,55]
```

Отметим, что бесконечный список `fib`, определяется с помощью самого себя.

Рассмотрим другие примеры бесконечных структур. Определим список из совершенных чисел.

```
-- список делителей числа n
factors :: Integer -> [Integer]
factors n    = [ i | i<-[1..n-1], n `mod` i == 0 ]

-- определение совершенного числа
perfect :: Integer -> Bool
perfect n    = sum (factors n) == n

-- список совершенных чисел
perfects :: [Integer]
perfects     = filter perfect [1..] -- встроенная функция filter выделяет из
                                    -- списка элементы с указанным свойством
> take 3 perfects
[6,28,496]
```

Определим бесконечный список простых чисел с помощью решета Эратосфена.

```
primes  :: [Integer]
```

---

[2] Форма [<элемент>|<элемент> <– <список>, <логическое условие>] играет в языке Haskell такую же роль, как определение множества элементов, удовлетворяющих некоторому свойству; обычная запись в математике имеет вид {x | x∈X & P(x)}. Использование форм позволяет создавать ясные и короткие программы. Пример «быстрой сортировки»:

```
quicksort :: [Char] -> [Char]
quicksort [] = []
quicksort (x:xs) = quicksort [y | y <- xs, y <= x] ++
                   [x] ++
                   quicksort [y | y <- xs, y > x]

> quicksort "Why use Haskell?"
"  ?HWaeehkllssuy"
```

```
primes = sieve [ 2.. ]
         where
         sieve (p:x) = p : sieve [ n | n <- x, n `mod` p /= 0 ]

> take 20 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

10.4 О модульном программировании

Мы убеждены, что модульный подход является ключом к успешному программированию.

Модульное программирование – это такой способ программирования, при котором вся программа разбивается на группу компонентов, называемых модулями, причем каждый из них имеет свой контролируемый размер, четкое назначение и детально проработанный интерфейс с внешним миром. Единственная альтернатива модульности – монолитная программа, что, конечно, неудобно. Таким образом, наиболее интересный вопрос при изучении модульности – определения критерия разбиения на модули. В основе модульного программирования лежат три основных принципа [7, с. 232]

.

- *Принцип утаивания информации* Парнаса. Всякий компонент утаивает единственное проектное решение, т.е. модуль служит для утаивания информации. Подход к разработке программ заключается в том, что сначала формируются список проектных решений, которые особенно трудно принять или которые, скорее всего, будут меняться. Затем определяются отдельные модули, каждый из которых реализует одно из указанных решений.
- *Аксиома модульности* Коуэна. Модуль – независимая программная единица, служащая для выполнения некоторой определенной функции программы и для связи с остальной частью программы.
- *Сборочное программирование* Цейтина. Модули – это программные кирпичи, из которых строятся программа. Существует три основные предпосылки к модульному программированию:
  - стремление к выделению независимой единицы программного знания. В идеальном случае всякая идея (алгоритм) должна быть оформлена в виде модуля;
  - ответ на вызов сложности программирования;
  - возможность параллельного исполнения модулей.

Языки, от которых требуется эффективность, должны хорошо поддерживать модульность. Но новых правил видимости и механизма для раздельной компиляции недостаточно – модульность подразумевает больше, чем просто использование модулей. Наши возможности разбиения задачи на подзадачи прямо зависят от наших возможностей соединения («склеивания») решений подзадач вместе. Чтобы способствовать модулярному программированию языки должны содержать хороший «клей». Языки функционального программирования обеспечивают два новых вида клея – **функции высокого порядка** и **ленивые вычисления**. Используя эти возможности для склеивания, каждый может модуляризовать программу новыми и перспективными путями. Небольшие и очень общие модули могут неоднократно широко использоваться, путем очень простого последовательного программирования. Это объясняет, почему функциональные программы так малы по размерам и легки в написании по сравнению с привычными императивными программами. Если какая-то часть программы выглядит беспорядочной или сложной, программисту следует попытаться модуляризовать ее и обобщить части программы. Он может ожидать, что функции высокого порядка и ленивые вычисления помогут это сделать. Более подробно об этих вопросах см. [13].

10.5 Надежность программирования

Рассмотрим вопросы надежности функционального программирования. Полезно ошибки выполнения программы разделить на два класса.

- **Класс А**: ошибки, приводящие к немедленному прерыванию работы программы. Примерами таких ошибок является деление на ноль или доступ по неправильному адресу.
- **Класс В**: ошибки, не вызывающие немедленного прекращения работы программы. К таким ошибкам относятся выход за пределы массива или незамеченная неправильная адресация.

Языки программирования можно разделить на следующие два класса.

- **Типизированные языки (языки с контролем типов).** Язык является типизированным, если существует система типов для него, независимо от того присутствуют действительно или нет типы в синтаксисе программы. Языки с контролем типов являются *явно типизированные*, если типы есть часть синтаксиса, и *неявно типизированные* в противном случае. Полностью неявно типизированные языки являются редкостью, но такие языки как Haskell и Clean поддерживает написание больших программных фрагментов с опущенной информацией о типах; система типов в этих языках автоматически выводит тип для таких программных фрагментах.
- **Нетипизированные языки.** Языки, в которых не ограничены множества значений переменных, является нетипизированными: они не имеют типов, или, эквивалентно, имеют один универсальный тип, которому принадлежат все используемые величины. В этих языках операции могут применяться к неподходящим аргументам: результат может иметь вполне определенное значение или операция может оканчиваться неудачей или приводить к непредсказуемым последствиям. Чистое ламбда–исчисление является предельным случаем нетипизированных языков: в нем единственная операция – аппликация и она всегда успешна, поскольку любая величина есть функция.

Программный фрагмент называется *надежным* (или безопасным), если он при своем выполнении не приводит к ошибке класса В. Языки, в которых все программные фрагменты надежны, называются *надежными* языками. Поэтому надежные языки исключают большинство коварных, незаметно появляющихся ошибок.

- Нетипизированные языки могут быть надежными за счет проверок во время выполнения.
- Типизированные языки могут быть надежными, если до выполнения отвергаются все потенциально ненадежные программы. Кроме того, в этих языках может присутствовать проверка во время работы программы. Главная цель системы типов языка – обеспечение надежности языка за счет исключения всех ошибок класса В.

Большинство нетипизированных языков есть, по необходимости, полностью надежными (например, Лисп). Иначе программирование в отсутствии проверок во времени компиляции и во времени работы программы приводило бы к весьма ненадежным программам. Ассемблер принадлежит к неприятной категории нетипизированных ненадежных языков.

Надежность – несколько примеров

	Типизированные	**Нетипизированные**
Надежные	Haskell, Clean	Lisp
Ненадежные	C	Assembler

Как правило, отсутствие надежности в разработке программ мотивируют временем исполнения кода. Проверки времени выполнения, необходимые для того, чтобы сделать программу безопасной, дорого стоят. Надежность требует стоимостных затрат даже в языках с интенсивным статическим анализом: тестирование выхода за пределы массива невозможно полностью сделать во время компиляции.

Стоимость безопасности оправдывается эффективностью программирования. Надежность обеспечивает остановку работы программы в случае ошибки выполнения и тем самым уменьшает время отладки. Более того, надежность гарантирует целостность структур данных во время выполнения программы и поэтому дает возможность собрать (очистить) мусор. В свою очередь, чистка мусора значительно уменьшает размер кода, и, тем самым, в некоторых случаях уменьшает стоимость выполнения.

Поэтому выбор между надежным и ненадежным языком может быть, в конечном счете, сведен к изменению пропорций между временем разработки программы и временем выполнения кода. Хотя, несомненно, достоинства безопасности еще не привели к широкому распространению надежных языков. Вместо того чтобы считать, что отсутствие надежности является злом, многие разработчики привыкают к мысли о ненадежности программирования.

В реальности некоторые статически проверяемые языки не являются достаточно надежными. Эти языки, используя эвфемизм, называют языками со *слабым контролем* (в литературе встречается также термин *слабо типизированый язык*), означающий, что некоторые ошибки класса B обнаруживаются во время компиляции, но не все. Такие языки широко варьируются по своей слабости контроля.

Например, в Паскале ненадежными являются только записи с вариантами и параметры подпрограмм без типа, в то время как в C присутствуют и широко используются многие ненадежные особенности, такие как арифметика указателей и приведение типов. Многие из проблем, вызванных слабым контролем, несколько смягчены в C++, и еще более в языке Java, подтверждая тем самым тенденцию отказа от слабого контроля. Язык Modula-3 поддерживает опасные черты, но только в модулях, которые явно объявляются ненадежными, и защищает надежные модули от импорта ненадежного интерфейса.

Подведем резюме. *Весь процесс разработки и сопровождения программного продукта на чистом и ленивом функциональном языке осуществляется эффективно и позволяет создавать надежные программы*.

Литература

1. Барендрегт Х. Ламбда–исчисление. Его синтаксис и семантика. – М.: Мир, 1985.–606с.
2. Грэй П. Логика, алгебра и базы данных: Пер. с англ.– М.: Машиностроение, 1989. – 359с.
3. Дейкстра Э. Заметки по структурному программированию // Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М.: Мир, 1975, с.7–97.
4. Математическая логика в программировании: Сб. статей 1980-1988 гг.: Пер. с англ. – М.: Мир, 1991.– 408с.
5. Мендельсон Э. Введение в математическую логику – М.: Наука, 1976.– 320с.
6. Непейвода Н. Н. Прикладная логика: Учебное пособие. – Новосибирск, Изд–во Новосиб. ун–та, 2000. – 521 с.
7. Одинцов И. О. Профессиональное программирование. Системный подход. – СПб.: БХВ–Петербург, 2002. – 512 с.
8. Филд А., Харрисон П. Функциональное программирование. – М.: Мир, 1993. – 637 с.
9. Хофштадтер Д. Гёдель, Эшер, Бах: эта бесконечная гирлянда. – Самара: Издательский Дом «Бахрах-М», 2001. – 752 с.
10. Энгелер Э. Метаматематика элементарной математики. – М.: Мир,1987. – 128 с.
11. Язык Пролог в пятом поколении ЭВМ: Сб. статей 1983 – 1986 гг. / Сост. Н. И. Ильинский. – М.: Мир, 1988.
12. Church A. The Calculi of Lambda Conversion. Princeton University Press, Princeton, 1941.
13. Hughes J. Why Functional Programming Matters. In D. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.
14. Lethbridge T. C. Priorieties for education and training of software engineers. // The Jornal of Systems and Software, 53, 200, pp. 53-57.
15. Schönfincel M. Über die Bausteine der mathematischen Logik. Math. Annalen, 92, 1924, s. 305–316.