

Введение в функциональное программирование

John Harrison
`jrh@cl.cam.ac.uk`

3rd December 1997

Предисловие

Эта книга основана на конспектах лекций по курсу *Введение в функциональное программирование*, который я читал в университете Кембриджа в 1996/7 учебном году.

Курс читался студентам последнего курса и аспирантам первого года обучения. К тому времени они уже свободно владели императивным программированием на языке Modula-3, а параллельно с этим курсом им читался курс по С. Этот курс определённо является курсом по функциональному программированию, а не курсом по программированию с использованием функциональных языков. Я попытался сделать упор на необычные свойства функциональных языков программирования, а также показать естественную область применения языка ML. Я не рассматриваю много важных аспектов, которые не присущи функциональному программированию, например абстрактные типы. В разделах помеченных звёздочкой размещён дополнительный материал, который не входит в экзамены, и который можно пропустить без потери понимания курса. Однако, я надеюсь, что некоторые заинтересуются более глубоким изучением отдельных частей курса и эти люди убедятся, что разделы, помеченные звёздочкой, являются естественным дополнением к основному тексту.

В предыдущие годы этот курс преподавался Майком Гордоном (Mike Gordon). Я сохранил структуру его курса, которая является смесью теории и практики, и была заимствована из его собственных лекционных материалов, доступных в части Part II книги ([Gordon 1988](#)). На меня также оказали сильное влияние те люди, которые читали здесь сопутствующие курсы – Andy Gordon, Larry Paulson, и, в главе о типах – курс, созданный Andy Pitts.

Большая глава с примерами не требуется на экзаменах. Она предназначена для улучшения понимания предыдущих частей, а также для того, чтобы показать как ML может быть использован.

Большая часть глав включает примеры, либо созданные специально для данного курса, либо заимствованные из других курсов. Обычно они требуют некоторого размышления, вместо того, чтобы быть стандартными задачами. Те, которые я посчитал сложными, отмечены знаком (*).

Эти материалы не были интенсивно протестированы, и без сомнения содержат различные ошибки и неясности. Я буду благодарен за конструктивную критику со стороны любого читателя.

John Harrison (jrh@cl.cam.ac.uk).

План лекций

Этот раздел описывает как материал распределён по 12 лекциям курса, каждая из которых длится немногим меньше часа.

1. **Введение и обзор** Функциональное и императивное программирование: различия, «за» и «против». Общая структура курса: как λ -исчисление превратилось в язык программирования общего назначения. λ -нотация: как разъясняет связывание переменных и как предоставляет средства общего анализа математической записи. Каррирование. Парадокс Рассела.
2. **λ -исчисление как формальная система** Свободные и связанные переменные. Подстановка. Правила преобразования. Эквивалентность λ -термов. Экстенциональность. Редукция и её стратегии. Теорема Чёрча-Россера: формулировка и следствия. Комбинаторы.
3. **λ -исчисление как язык программирования** Становление теории алгоритмов; полнота по Тьюрингу (без доказательства). Представление данных и основные операции: логические значения, пары и кортежи, натуральные числа. Вычисление предшествующего числа. Определение рекурсивных функций: комбинаторы неподвижной точки. Let-выражения. λ -исчисление как декларативный язык.
4. **Типы** Зачем нужны типы? Ответы из программирования и логики. Простое типизированное λ -исчисление. Типизация по Чёрчу и Карри. Let-полиморфизм. Наиболее общие типы и алгоритм Милнера. Сильная нормализация (без доказательства), и её негативное влияние на полноту по Тьюрингу. Добавляем оператор рекурсии.
5. **ML** ML как типизированное λ -исчисление с энергичным вычислением. Подробности стратегии вычисления. Условное выражение. Семейство языков ML. Практика работы с ML. Создание функций. Связывания и объявления. Рекурсивные и полиморфные функции. Сравнение функций.
6. **Более подробно о ML** Загрузка кода из файлов. Комментарии. Основные типы данных: процедурный, логические, числа и строки. Встроенные операции. Конкретный синтаксис и инфиксные операции. Дополнительные примеры. Рекурсивные типы и сопоставление с образцом. Примеры: списки и рекурсивные функции для работы со списками.
7. **Доказательство корректности программ** Проблема корректности. Тестирование и верификация. Область применимости верификации.

Функциональные программы как математические объекты. Примеры доказательства свойств программ: вычисление степени и НОД, конкатенация и обращение списков.

8. **Эффективный ML** Использование стандартных комбинаторов. Проход по списку и другие полезные примеры использования комбинаторов. Хвостовая рекурсия и аккумуляторы; почему хвостовая рекурсия более эффективна. Принудительное вычисление. Минимизация операций cons. Более эффективная реализация обращения данных. Использование «as». Императивные возможности: исключения, ссылки, массивы и последовательность вычислений. Императивные возможности и типы; ограничение значения.
9. **Примеры на ML I: символьное дифференцирование** Символьные вычисления. Представление данных. Приоритет операторов. Ассоциативные списки. Форматированный вывод выражений. Устанавливаем принтер. Дифференцирование. Упрощение. Проблема «правильного» упрощения.
10. **Примеры на ML II: синтаксический анализ** Понятие грамматики, задача синтаксического анализа. Устранение неоднозначностей. Метод рекурсивного спуска. Реализация синтаксического анализа на языке ML. Комбинаторы синтаксического анализа, примеры. Лексический анализ. Анализатор термов. Автоматический учёт приоритетов операций. Устранение возвратов. Сравнение с другими методами.
11. **Примеры на ML III: арифметика вещественных чисел** Вещественные числа и конечные представления. Вещественные числа как программы или функции. Выбор представления вещественных чисел. Целые числа произвольной разрядности. Преобразование целочисленных значений в вещественные. Операции смены знака и вычисления абсолютной величины. Сложение: важность деления с округлением. Умножение и деление на целое число. Умножение: общий случай. Обратные числа и деление. Отношения порядка и равенства. Тестирование. Устранение избыточных вычислений при помощи функций с памятью.
12. **Примеры на ML IV: Пролог и доказательство теорем** Выражения в Прологе. Лексический анализ с учётом регистра. Разбор и печать, включая списочный синтаксис. Унификация. Поиск с возвратом. Примеры выражений в Пролог. Доказательство теорем в стиле Пролог. Работа с формулами; отрицательная нормальная форма. Базовое средство доказательства; использование продолжений. Примеры: проблемы Пельетьера и программа, расследующая преступления.

Оглавление

1	Введение	1
1.1	Достоинства функционального программирования	3
1.2	План	5
2	Лямбда-исчисление	7
2.1	Преимущества лямбда-нотации	8
2.2	Парадокс Рассела	11
2.3	Лямбда-исчисление как формальная система	12
2.3.1	Лямбда-термы	12
2.3.2	Свободные и связанные переменные	13
2.3.3	Подстановка	14
2.3.4	Преобразования	16
2.3.5	Равенство лямбда-выражений	16
2.3.6	Экстенциональность	17
2.3.7	Лямбда-редукция	18
2.3.8	Стратегии редукции	18
2.3.9	Теорема Чёрча-Россера	19
2.4	Комбинаторы	21
3	Лямбда-исчисление как язык программирования	25
3.1	Представление данных в лямбда-исчислении	27
3.1.1	Логические значения и условия	27
3.1.2	Пары и кортежи	28
3.1.3	Натуральные числа	30
3.2	Рекурсивные функции	32
3.3	Let-выражения	33
3.4	Достижение уровня полноценного языка программирования	35
3.5	Дополнительная литература	36
4	Типы	37
4.1	Типизированное лямбда-исчисление	39
4.1.1	Множество допустимых типов	39
4.1.2	Типизация по Чёрчу и Карри	40
4.1.3	Формальные правила типизации	41
4.1.4	Сохранение типа	42
4.2	Полиморфизм	43
4.2.1	Проблемы let-полиморфизма	44
4.2.2	Наиболее общий тип	45

4.3	Сильная нормализация	47
5	Знакомство с ML	51
5.1	Энергичное вычисление	51
5.2	Результаты энергичного вычисления	54
5.3	Семейство языков ML	55
5.4	Запуск ML	55
5.5	Взаимодействие с ML	56
5.6	Связывания и объявления	57
5.7	Полиморфные функции	59
5.8	Равенство функций	60
6	Более подробно о ML	63
6.1	Основные типы данных и операции	64
6.2	Дальнейшие примеры	67
6.3	Определения типов	69
6.3.1	Сопоставление с образцом	70
6.3.2	Рекурсивные типы	73
6.3.3	Древовидные структуры	75
6.3.4	Тонкости рекурсивных типов	77
7	Доказательство корректности программ	81
7.1	Функциональные программы как математические объекты	83
7.2	Вычисление степени	84
7.3	Вычисление НОД	85
7.4	Конкатенация списков	87
7.5	Обращение списков	88
8	Эффективный ML	95
8.1	Полезные комбинаторы	95
8.2	Создание эффективного кода	97
8.2.1	Хвостовая рекурсия и аккумуляторы	97
8.2.2	Минимизация операций cons	99
8.2.3	Принудительное вычисление	102
8.3	Императивные возможности	103
8.3.1	Исключения	103
8.3.2	Ссылки и массивы	104
8.3.3	Последовательность вычислений	106
8.3.4	Работа с системой типов	106
9	Примеры	111
9.1	Символьное дифференцирование	111
9.1.1	Термы первого порядка	111
9.1.2	Печать	112
9.1.3	Дифференцирование	116
9.1.4	Упрощение	117
9.2	Синтаксический анализ	119
9.2.1	Метод рекурсивного спуска	121

9.2.2	Комбинаторы синтаксического анализа	122
9.2.3	Лексический анализ	124
9.2.4	Анализатор термов	125
9.2.5	Автоматический учёт приоритетов	126
9.2.6	Недостатки метода	128
9.3	Точная арифметика вещественных чисел	131
9.3.1	Выбор представления вещественных чисел	131
9.3.2	Целые числа произвольной разрядности	132
9.3.3	Основные операции	133
9.3.4	Умножение: общий случай	137
9.3.5	Обратные числа	138
9.3.6	Отношения порядка	140
9.3.7	Кэширование	141
9.4	Пролог и доказательство теорем	144
9.4.1	Термы пролога	145
9.4.2	Лексический анализ	145
9.4.3	Синтаксический анализ	146
9.4.4	Унификация	148
9.4.5	Перебор с возвратами	150
9.4.6	Примеры	151
9.4.7	Доказательство теорем	152

Глава 1

Введение

Программы, написанные на традиционных языках программирования, таких как FORTRAN, Algol, C и Modula-3, в своей работе опираются на изменение значений набора переменных, называемого *состоянием*. Если мы пренебрежём операциями ввода-вывода и вероятностью того, что программа будет работать постоянно (например, управляющая система для производства), то мы можем прийти к следующей абстракции. Первоначально состояние имеет некоторое значение σ , представляющее собой входные данные для программы, а после завершения её исполнения — новое значение σ' , представляющее результаты. Выполнение отдельных операторов сводится к изменению ими состояния, которое последовательно проходит через конечное число значений:

$$\sigma = \sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \dots \rightarrow \sigma_n = \sigma'$$

Например, в программе сортировки состояние первоначально включает в себя массив значений, а после того, как программа завершается, состояние модифицируется таким образом, что эти значения становятся упорядоченными, в то время как промежуточные состояния представляют собой ход достижения данной цели.

Состояние обычно изменяется с помощью операторов *присваивания*, часто записываемых в виде $v = E$ или $v := E$, где v — переменная, а E — некоторое выражение. Последовательность выполнения таких операторов задаётся в тексте программы их размещением друг за другом (при этом часто в качестве разделителя применяется точка с запятой). С помощью составных операторов, таких как `if` и `while`, можно выполнять операторы в зависимости от условия или циклически, часто полагаясь на другие свойства текущего состояния. В результате программа превращается в набор инструкций по изменению состояния, и поэтому данный стиль программирования часто называется *императивным* или *процедурным*. Соответственно, традиционные языки программирования, поддерживающие такой стиль, также известны как императивные или процедурные языки.

Функциональное программирование радикально отличается от этой модели. По существу, функциональная программа представляет собой просто выражение, а выполнение программы — процесс его вычисления.¹ В общих чертах мы можем понять, как это возможно, используя следующие рассуждения. Предположим, что

¹Функциональное программирование часто называют «аппликативным программированием», поскольку основной его механизм — это *аппликация* (*применение*) функции к аргументам.

императивная программа (вся целиком) детерминирована, т.е. выход полностью определяется входом; мы можем сказать, что конечное состояние или тот его фрагмент, который нас интересует, являются функцией начального состояния, например $\sigma' = f(\sigma)$.² В функциональном программировании эта точка зрения имеет особое значение: программа – это выражение, которое соответствует математической функции f . Функциональные языки поддерживают создание таких выражений за счет того, что позволяют использовать мощные функциональные конструкции.

Функциональное программирование может противопоставляться императивному как с хорошей, так и в плохой стороны. К недостаткам ФП можно отнести то, что функциональные программы не используют переменные – то есть *не имеют* состояния. Соответственно, они не могут использовать присваивание, поскольку нечему присваивать. Кроме того, идея последовательного выполнения операторов также бессмысленна, поскольку первый оператор не имеет никакого влияния на второй, так как нет никакого состояния, передаваемого между ними. К достоинствам функционального подхода можно отнести то, что функциональные программы могут использовать функции более изящным способом. Функции могут рассматриваться точно так же, как и более простые объекты, такие как целые числа: они могут передаваться в другие функции как аргументы и возвращаться в качестве результатов, а также применяться в вычислениях. Вместо последовательного выполнения операторов и использования циклов, функциональные языки программирования предлагают рекурсивные функции, т.е. функции, определённые в терминах самих себя. Большинство традиционных языков программирования обеспечивают весьма скудные возможности в этих областях. Язык C имеет некоторые ограниченные возможности работы с функциями при помощи указателей, но не позволяет создавать новые функции динамически, а язык FORTRAN вообще не поддерживает рекурсию.

Продемонстрируем разницу между императивным и функциональным программированием на примере функции вычисления факториала. Она может быть записана императивно на языке C как:

```
int fact(int n)
{ int x = 1;
  while (n > 0)
  { x = x * n;
    n = n - 1;
  }
  return x;
}
```

в то время как на языке ML (функциональном языке программирования, который мы обсудим позже) она может быть реализована в виде рекурсивной функции:

```
let rec fact n =
  if n = 0 then 1
  else n * fact(n - 1);;
```

Можно отметить, что такое определение достаточно просто реализовать и

²Сравните это с замечанием Наура ([Raphael 1966](#)) о том, что мы можем записать любую программу в виде одного выражения $Output = Program(Input)$.

на языке С. Однако, при необходимости более сложной работы с функциями функциональные языки не имеют себе равных.

1.1 Достоинства функционального программирования

На первый взгляд, язык без переменных или возможности последовательного выполнения инструкций кажется совершенно непрактичным. Это впечатление не может быть разрушено с помощью нескольких слов, написанных тут. Но мы надеемся, что изучая материал, приведённый далее, читатель получит представление о том, какое разнообразие задач можно решить, программируя в функциональном стиле.

Императивный стиль программирования не является нерушимой догмой. Многие свойства императивных языков программирования развились в процессе абстрагирования от типового компьютерного оборудования, от машинного кода к ассемблерам, затем к макроассемблерам, языку FORTRAN и так далее. Нет оснований утверждать, что такие языки представляют собой наиболее удобный способ взаимодействия человека и машины. В самом деле, последнее слово в развитии компьютерных архитектур еще не сказано, и компьютеры должны служить нашим нуждам, а не наоборот. Вероятно, что правильный подход не в том, чтобы начать с оборудования и продвигаться вверх, а наоборот – начать работу с языка программирования, как средства для описания алгоритмов, и затем двигаться *вниз* к оборудованию (Dijkstra 1976). В действительности, данная тенденция может быть обнаружена и в традиционных языках программирования. Даже FORTRAN позволяет записывать арифметические выражения обычным способом. Программист не обеспокоен задачей линеаризации вычисления подвыражений и выделения памяти для хранения промежуточных результатов.

Из этих соображений можно сделать вывод, что идея разработки языков программирования, сильно отличающихся от традиционных, императивных языков, является вполне законной. Однако, для того, чтобы показать, что мы не просто предлагаем изменения ради изменений, мы должны сказать несколько слов о том, почему мы могли бы предпочесть функциональные языки программирования императивным.

Возможно, главной причиной является то, что программы на функциональных языках более точно соответствуют математическим объектам, и их свойства легче доказывать. Для того, чтобы показать, что программа означает, мы можем связать абстрактный математический смысл с программой или оператором — это цель *денотационной семантики* (семантика = значение, смысл). В императивных языках это должно делаться скорее побочным способом, из-за неявной зависимости от состояния. Для простых императивных языков можно связать оператор с функцией $\Sigma \rightarrow \Sigma$, где Σ – множество допустимых состояний. Таким образом, оператор получает некоторое состояние и порождает другое. Однако, не каждый оператор всегда завершает свою работу (например, `while true do x := x`), так что эта функция, вообще говоря, является частичной. Иногда более предпочтительными являются альтернативные средства формализации семантики, например, *преобразователи предикатов* (Dijkstra 1976). Но если мы добавим возможности, которые могут

сложным образом изменить последовательность выполнения операторов, например, `goto`, или конструкции `break` и `continue` языка C, то даже такие решения перестанут работать, поскольку один оператор может привести к пропуску выполнения других операторов, следующих за ним в тексте программы. Вместо этого обычно используют более сложные семантики, основанные на *продолжениях* (*continuations*).

В противоположность сказанному выше, функциональные программы, по словам Хенсона [Henson \(1987\)](#), «носят свою семантику с собой».³ Мы можем показать это на примере ML. Основные типы напрямую могут рассматриваться как математические объекты. Используя стандартную запись $\llbracket X \rrbracket$ для «семантики X », мы можем сказать, например, что $\llbracket \text{int} \rrbracket = \mathbb{Z}$. Например, функция ML `fact`, определённая выражением:

```
let rec fact n =
  if n = 0 then 1
  else n * fact(n - 1);;
```

имеет один аргумент типа `int`, и возвращает значение типа `int`, так что она просто является связанной с абстрактной частичной функцией $\mathbb{Z} \rightarrow \mathbb{Z}$:

$$\llbracket \text{fact} \rrbracket(n) = \begin{cases} n! & \text{if } n \geq 0 \\ \perp & \text{otherwise} \end{cases}$$

(Здесь \perp обозначает неопределённость, поскольку для отрицательных аргументов программа не сможет завершиться). Однако этот способ простой интерпретации не работает для не-функциональных программ, поскольку, так называемые «функции» могут не быть функциями в математическом смысле. Например, в стандартной библиотеке языка C есть функция `rand()`, которая возвращает различные псевдослучайные значения при последовательных вызовах. Это может быть сделано с помощью локальных статических переменных, используемых для хранения предыдущих результатов, например, так:

```
int
rand(void) {
    static int n = 0;
    return n = 2147001325 * n + 715136305;
}
```

Таким образом, мы можем рассматривать отказ от переменных и присваивания, как следующий шаг после отказа от `goto`, поскольку каждый следующий шаг делает семантику проще. Более простая семантика делает доказательство свойств программ более ясным. Это даёт нам больше возможностей для доказательства корректности программ и преобразований, используемых для их оптимизации.

У функциональных языков есть и другое потенциальное преимущество. Поскольку вычисление выражений не имеет побочных эффектов для любых состояний, то отдельные подвыражения могут вычисляться в произвольном порядке, не влияя друг на друга. Это означает, что функциональные программы хорошо поддаются распараллеливанию, т.е., компьютер может автоматически вычислять различные подвыражения на разных процессорах. В то же время, императивные программы часто задают жёсткий порядок вычислений, так что даже ограниченное

³Дополнительно: денотационную семантику можно рассматривать как попытку превратить императивные языки в функциональные, путём явного объявления состояний.

перемешивание инструкций в современных процессорах с конвейерной обработкой ведёт к сложностям и возникновению технических проблем.

На самом деле, ML не является чисто функциональным языком программирования; в нём есть переменные и присваивания, если потребуется. Большую часть времени мы будем делать нашу работу, оставаясь в рамках чисто функционального подмножества языка. Но даже если мы и воспользуемся присваиваниями, потеряв некоторые из ранее перечисленных достоинств, останется в силе большая гибкость в работе с функциями, свойственная языкам, подобным ML. Программы часто могут быть выражены очень кратко и элегантно при помощи функций высшего порядка (функций, которые оперируют другими функциями).⁴ Код может быть более общим, поскольку он может быть параметризован другими функциями. Например, программа, которая складывает список чисел, и программа, которая умножает список чисел, могут рассматриваться как экземпляры одной и той же программы, которая параметризуется арифметической операцией над парой чисел и единичным элементом. В первом случае это будут $+$ и 0 , а во втором $-$ $*$ и 1 .⁵ В заключение, функции могут использоваться для представления *бесконечных* наборов данных удобным способом, например, мы позже покажем как использовать функции для выполнения вычислений с вещественными числами, в отличие от использования приближений в виде чисел с плавающей запятой.

В то же время, функциональные программы не лишены собственных проблем. Поскольку функциональные программы менее ориентированы на выполнение на конечном оборудовании, может быть тяжело вычислить точное использование ресурсов, таких как время и память. Ввод-вывод также непросто ввести в функциональную модель, хотя существуют остроумные способы, основанные на бесконечных последовательностях.

Читатели данной книги должны сами сделать заключение о достоинствах функционального стиля. Мы не хотим навязывать никакую идеологию, а лишь хотим показать, что *существуют* разные подходы к программированию, и что в соответствующих ситуациях функциональное программирование может иметь значительные достоинства. Большинство наших примеров выбрано из областей, которые могут быть определены как «символьные вычисления». Мы верим, что функциональные программы успешно работают в таких приложениях. Однако, как всегда, человек должен использовать наиболее подходящие для работы инструменты. Возможно, что императивное, объектно-ориентированное или логическое программирование лучше подходят для определённых задач.

1.2 План

Для тех, кто использовал императивное программирование, переход к функциональному будет неизбежно тяжёл, независимо от используемого подхода. Хотя есть люди, которые сразу хотят перейти непосредственно к программированию,

⁴Элегантность субъективна, а краткость — не самоцель. Функциональные языки, а также другие языки, такие как APL, часто создают соблазн создания очень короткого, хитроумного кода, который элегантен для знатоков, но непонятен для остальных.

⁵Это напоминает абстракции, введенные в чистой математике; например, аддитивные и мультипликативные структуры над числами являются частными случаями абстрактного понятия «моноид». Такое подобие помогает избегать дублирования и увеличивает элегантность.

мы выбрали другой подход – мы начнём с λ -исчисления, и покажем как оно может быть использовано в роли теоретической основы функциональных языков. Такой подход обладает тем достоинством, что он хорошо соответствует реальной истории разработки функциональных языков.

Следовательно, сначала мы введём λ -исчисление, и покажем как оно, первоначально предназначенное на роль формальной логической основы математики, превратилось в полноценный язык программирования. Затем мы обсудим, зачем мы хотим добавить типы в λ -исчисление, и покажем, как добиться требуемого. Это приведёт нас к языку ML, который по существу является оптимизированной реализацией типизированного λ -исчисления с определённой стратегией вычисления выражений. Мы рассмотрим практические основы функционального программирования на ML, обсудим полиморфизм и понятие наиболее общего типа данных. Затем мы перейдём к более сложным темам, таким как исключения и императивные возможности ML. В заключение, мы приведём несколько реальных примеров, которые, как мы надеемся, подтвердят мощь ML.

Дополнительная литература

Множество книг о «функциональном программировании» включают в себя общее введение и описание отличий от императивного программирования — просмотрите несколько и выберите ту, которая вам нравится. Например, [Hen-son \(1987\)](#) предлагает хорошее вводное обсуждение и содержит такую же смесь теории и практики, как и в этом тексте. Детальная и спорная пропаганда функционального стиля программирования приведена в работе создателя FORTRAN Джона Бэкуса [Backus \(1978\)](#). Работа [Gordon \(1994\)](#) обсуждает проблемы введения ввода-вывода в функциональные языки, а также приводит некоторые решения. Читатели, заинтересовавшиеся денотационной семантикой для императивных и функциональных языков, могут прочитать [Winskel \(1993\)](#).

Глава 2

λ -исчисление

λ -исчисление основывается на так называемой ‘ λ -нотации’ для обозначения функций. В неформальной математике, когда кто-то хочет сослаться на функцию, то обычно сначала дает этой функции произвольное имя, а затем использует его, например:

Предположим, что $f : \mathbb{R} \rightarrow \mathbb{R}$ определена выражением:

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ x^2 \sin(1/x^2) & \text{if } x \neq 0 \end{cases}$$

Тогда $f'(x)$ не интегрируема по Лебегу в пределах $[0, 1]$.

Большинство языков программирования, например C, в этом отношении похожи: мы можем определить функции только давая им имена. Например, для того, чтобы использовать функцию `successor` (которая добавляет 1 к своим аргументам) не самым простым способом (например, используя указатель на нее), то хотя она и является очень простой, но все равно нам нужно назвать ее используя определение функции, такое как:

```
int suc(int n) {  
    return n + 1;  
}
```

В математике или программировании это кажется нормальным, и в общем случае, работает достаточно хорошо. Однако, это может стать неудобным, когда начинают использоваться функции высшего порядка (функции, которые манипулируют другими функциями). В любом случае, если мы хотим рассматривать функции наравне с другими математическими объектами, то требование именования функций будет нелогичным. При обсуждении арифметического выражения, построенного из более простых выражений, мы просто записываем подвыражения не давая им имен. Представим, что если бы мы всегда работали с арифметическими выражениями таким способом:

Определим x и y , так что $x = 2$ и $y = 4$. Тогда $xx = y$.

λ -нотация позволяет записывать функции практически тем же способом, что и остальные виды математических объектов. Существует общепринятая нотация,

которая иногда использовалась в математике для этих целей, хотя обычно она используется как часть определения временного имени. Мы можем записать

$$x \mapsto t[x]$$

для обозначения функции, отображающей любой аргумент x в некоторое произвольное выражение $t[x]$, которое обычно, но не обязательно, содержит x (иногда полезно “отбросить” аргумент). Однако, мы должны использовать другую нотацию, разработанную Church (1941):

$$\lambda x. t[x]$$

которое должно читаться точно также, как и предыдущее выражение. Например, $\lambda x. x$ является функцией отображения (identity function), которая просто возвращает переданный аргумент, в то время как $\lambda x. x^2$ является функцией возведения в квадрат.

Выбор символа λ является произвольным, и не несет никакой смысловой нагрузки. (Можно часто видеть, особенно во французских текстах, альтернативную нотацию $[x] \ t[x]$.) Вероятно, что она возникла во время сложного процесса эволюции. В начале, в известной книге *Principia Mathematica* (Whitehead and Russell 1910) использовалась ‘hat’-нотация $t[\hat{x}]$ для функции от x производящей $t[x]$. Чёрч (Church) изменил его на $\hat{x}. t[x]$, но поскольку наборщики текстов не могли поместить значок ‘hat’ (крыша) над x , то оно появилось как $\wedge x. t[x]$, которое затем трансформировалось в $\lambda x. t[x]$ в руках других наборщиков.

2.1 Преимущества λ -нотации

Используя λ -нотацию мы можем прояснить некоторые неточности, внесенные неформальной математической нотацией. Например, часто говорят о ‘ $f(x)$ ’, используя контекст для определения того, что мы рассматриваем – саму функцию f , или результат ее применения к конкретному x . Дополнительной пользой будет то, что λ -нотация дает нам возможности анализа почти всей математической нотации. Если мы начнем с переменных и констант, и построим выражение используя лишь λ -абстракцию и применение функций к аргументом, то мы сможем представлять очень сложные математические выражения.

Мы будем использовать общепринятую нотацию $f(x)$ для операции применения функции f к аргументу x , за тем исключением, что в традиционной λ -нотации, скобки могут быть опущены, что позволяет нам записывать это выражение в виде $f x$. По причинам, которые станут понятны в процессе чтения следующего параграфа, мы считаем, что применение функции ассоциативно слева, т.е. $f x y$ означает $(f(x))(y)$. В качестве сокращенной записи для $\lambda x. \lambda y. t[x, y]$ мы будем использовать $\lambda x y. t[x, y]$, и т.д. Мы также предполагаем, что область λ -абстракции распространяется вправо, насколько это возможно. Например, $\lambda x. x y$ означает $\lambda x. (x y)$, а не $(\lambda x. x) y$.

На первый взгляд нам необходимо введение специальной нотации для функций нескольких аргументов. Однако существует способ перевода таких функций на обычную λ -нотацию. Этот способ называется *каррированием* (currying), по имени математика-логика Curry (1930). (В действительности, этот способ уже

использовался и Frege (1893), и Schönfinkel (1924), но легко понять, почему соответствующие применения не получили общественного призвания.) Идея заключается в использовании выражений, вида $\lambda x y. x + y$. Это выражение может рассматриваться как функция $\mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$, так что можно сказать, что оно является ‘функцией высшего порядка (higher order function)’ или ‘функционалом (functional)’, поскольку при применении к другой функции, она производит другую функцию, которая получает второй аргумент. На самом деле, она получает аргументы по очереди, по одному, а не все сразу. Например, рассмотрим:

$$(\lambda x y. x + y) 1 2 = (\lambda y. 1 + y) 2 = 1 + 2$$

Заметьте, что в λ -нотации, применение функции считается лево-ассоциативной операцией, поскольку каррирование используется очень часто.

λ -нотация в частности полезна в обеспечении унифицированной обработки связанных переменных. В математике переменные обычно выражают зависимость некоторого выражения от значения этой переменной; например, значение $x^2 + 2$ зависит от значения x . В таком контексте, мы будем говорить, что переменная является *свободной*. Однако, существуют другие ситуации, где переменная просто используется в качестве обозначения, а не показывает зависимость от значения. В качестве примеров можно рассмотреть переменную m в выражении

$$\sum_{m=1}^n m = \frac{n(n+1)}{2}$$

и переменную y в выражении

$$\int_0^x 2y + a \, dy = x^2 + ax$$

В логике, квантификаторы $\forall x. P[x]$ (‘для всех x , $P[x]$ ’) и $\exists x. P[x]$ (‘существует такой x так что $P[x]$ ’) являются дополнительными примерами, а в теории множеств, мы имеем абстрактные множества, наподобие $\{x \mid P[x]\}$, а также индексированные объединения и пересечения. В таких случаях говорят, что переменная должна быть *связанной (bound)*. В определенных подвыражениях она является свободной, но в полном выражении, она связана *операцией связывания переменных*, такой как сложение. Часть, находящаяся ‘внутри’ этой операции связывания переменных называется *областью видимости (scope)* связанной переменной.

Аналогичная ситуация возникает в большинстве языков программирования, по крайней мере, в произошедших от Algol 60. Переменные имеют определенную область видимости, а формальные аргументы процедур и функций, являются связанными переменными, например, n в определении на языке C функции `successor`, данном выше. Кто-то может рассматривать объявления переменных как операцию связывания для вложенных объектов соответствующих переменных. Между прочим, заметьте, что *область видимости* переменной должна быть отделена от ее *времени жизни*. В функции `rand` языка C, которую мы привели в введении, переменная n имеет ограниченную область видимости, но сохраняет свое значение, даже за пределами данного блока кода.

Мы можем свободно изменить имя связанной переменной, без изменения смысла выражения. Например,

$$\int_0^x 2z + a \, dz = x^2 + ax$$

Аналогичным образом, при использовании λ -нотации выражения $\lambda x. E[x]$ и $\lambda y. E[y]$ являются эквивалентными; это называется *альфа-эквивалентностью*, а процесс преобразования между такими парами, называется *альфа-конверсией*. Мы должны добавить оговорку, что y не является свободной переменной в выражении $E[x]$, иначе значение выражения изменится, например, как в

$$\int_0^x 2a + a \, da \neq x^2 + ax$$

Также возможно использовать в одном выражении одинаковые имена для свободных и связанных переменных; хотя это может сбивать с толку, но с технической точки зрения это не является неоднозначным, например

$$\int_0^x 2x + a \, dx = x^2 + ax$$

В действительности, обычная нотация Лейбница для производных имеет то же самое свойство, например, в:

$$\frac{d}{dx} x^2 = 2x$$

x используется и как связанная переменная для того, чтобы показать, что дифференцирование производится относительно x , и как свободная переменная, чтобы показать где будет происходить окончательное вычисление производной. Это может сбивать с толку, например, $f'(g(x))$ обычно означает что-то отличное от $\frac{d}{dx} f(g(x))$. Внимательные писатели, особенно в многомерных (FIXME multivariate) работах, часто явно показывают это отличие, записывая:

$$\left| \frac{d}{dx} x^2 \right|_x = 2x$$

или

$$\left| \frac{d}{dz} z^2 \right|_x = 2x$$

Одной из составляющей привлекательности λ -нотации является то, что все операции связывания переменных, такие как суммирование, дифференцирование и интегрирование, могут рассматриваться как функции, применяемые к λ -выражениям. Обобщение всех операций связывания переменных с помощью λ -абстракции, позволяет нам сконцентрироваться на технических проблемах связанных переменных в конкретной ситуации. Например, мы можем рассматривать $\frac{d}{dx} x^2$ как синтаксическую обвязку (syntactic sugaring) для $D (\lambda x. x^2) x$ где $D : (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \mathbb{R}$ является оператором дифференцирования, производящий производную первого аргумента (функции) в точке, указанной вторым аргументом. Преобразуя обычный синтаксис в λ -нотацию, мы получим $D (\lambda x. \text{EXP } x \, 2) x$ для некоторой константы EXP, представляющей экспоненциальную функцию.

Таким образом, λ -нотация очень привлекательна для математиков в качестве общего ‘абстрактного синтаксиса’; все что нам нужно – соответствующий набор констант с которыми мы начнем работать. λ -абстракция выглядит, ретроспективно,

как подходящий примитив в терминах которого проводится анализ связывания переменных. Эта идея уходит корнями к записи логики высшего порядка в λ -нотации, использованной Чёрчем, а также, как мы увидим в следующей главе, Landin указывал на то, как много конструкций в языках программирования имеет аналогичную интерпретацию. В последнее время, идея использования λ -нотации в качестве универсального абстрактного синтаксиса была введена Martin-Löf, и часто на нее ссылаются как на ‘теорию Martin-Löf для выражений и арности (arity)’.¹

2.2 Парадокс Рассела

Как мы уже упоминали, одной из привлекательных сторон λ -нотации является то, что она разрешает анализ почти всего математического синтаксиса. Первоначально, Чёрч надеялся продвинуться дальше, и затронуть теорию множеств, которая, как хорошо известно, является достаточно мощным средством для формирования основания для большей части современной математики. Взяв любое множество S , мы можем сформировать для него так называемый *параметрический предикат* (*characteristic predicate*) χ_S , такой, что:

$$\chi_S(x) = \begin{cases} true & \text{if } x \in S \\ false & \text{if } x \notin S \end{cases}$$

И наоборот, для любого унарного предиката (т.е. функции одного аргумента) P , мы можем рассмотреть множество всех x удовлетворяющих $P(x)$ — мы будем просто записывать $P(x)$ для $P(x) = true$. Таким образом, мы видим, что множества и предикаты являются лишь различными способами обсуждения одних и тех же вещей. Вместо рассмотрения S как множества, и записи $x \in S$, мы можем рассматривать его как предикат, и записывать как $S(x)$.

Это позволяет проведение обычного анализа в λ -нотации: мы можем допустить что произвольные λ -выражения являются функциями, и косвенным образом, и множествами. К сожалению, это приводит к несовместимости. Простейшим способом убедиться в этом, является рассмотрение парадокса Рассела про множество всех множеств, которые не содержат сами себя:

$$R = \{x \mid x \notin x\}$$

У нас имеется $R \in R \Leftrightarrow R \notin R$, сильное противоречие. В терминах функций, определенных с помощью λ -нотации, мы задаем $R = \lambda x. \neg(x x)$, а затем находим, что $R R = \neg(R R)$, рассчитывая на интуитивное понимание операции отрицания \neg .

Для того, чтобы избежать таких парадоксов, Church (1940) развивает идею Рассела путем добавления в λ -нотацию понятия *типа*; мы должны рассмотреть это в следующей главе. Однако сам парадокс предлагает некоторые интересные возможности в стандартной, не типизированной системе, в чем мы убедимся позже.

¹Она была представлена на симпозиуме в Brouwer в 1981 году, но не попала в печатные материалы симпозиума.

2.3 λ -исчисление как формальная система

Некоторые очевидные факты принимались нами без обоснования, например, что $(\lambda y. 1 + y) 2 = 1 + 2$, поскольку это соответствует желаемым свойствам операций применения и абстракции, которые считаются в определённом смысле взаимно обратными. Чтобы перейти от интуитивных действий к λ -исчислению, нам потребуется зафиксировать некоторые из основных принципов, объявив их (и *только* их) формальными правилами. Привлекательность этого шага в том, что правила в дальнейшем могут применяться механически, подобно тому, как преобразование уравнения $x - 3 = 5 - x$ в равносильное ему $2x = 5 + 3$ не требует каждый раз задумываться, *почему* допустимо такое перемещение слагаемых из одной части равенства в другую. Как писал Уайтхед в работе [Whitehead \(1919\)](#), формальная символика и правила действий...

[...] вводились всякий раз, когда требовались что-либо упростить.
[...] используя формальные обозначения, мы можем переходить от одного этапа рассуждений к другому почти механически, зрительно, в противном же случае нам пришлось бы задействовать гораздо больше интеллектуальных ресурсов. [...] Цивилизация прогрессирует, увеличивая количество важных операций, которые могут производиться, не задумываясь.

2.3.1 λ -термы

Основой λ -исчисления служит формальное понятие λ -термов, которые строятся из переменных и некоторого фиксированного множества констант при помощи операций применения (аппликации) функций и λ -абстракции. Это значит, что всевозможные λ -термы разбиваются на четыре класса:

1. **Переменные:** обозначаются произвольными алфавитно-цифровыми строками; как правило, мы будем использовать в качестве имен буквы, расположенные ближе к концу латинского алфавита, например, x , y и z .
2. **Константы:** количество констант определяется конкретной λ -нотацией, иногда их нет вовсе. Мы будем также обозначать их алфавитно-цифровыми строками, как и переменные, отличая их друг от друга по контексту.
3. **Комбинации:** применение функции s к аргументу t , где s и t представляют собой произвольные термы. Будем обозначать комбинации как $s t$, а их составные части называть «ратор» и «ранд» соответственно.²
4. **Абстракция** произвольного λ -терма s по переменной x (которая может как входить свободно в s , так и нет) имеет вид $\lambda x. s$.

Формально, этот набор правил представляет собой индуктивное определение множества λ -термов, т. е. последние могут конструироваться *только так*, как описано выше. Благодаря этому, мы получаем основания для

²Сокр. «оператор» и «операнд».

- определения функций над λ -термами при помощи примитивной рекурсии;
- доказательства утверждений о свойствах λ -термов методом структурной индукции.

Формальное изложение понятий индуктивного построения, а также примитивной рекурсии и структурной индукции доступно из многих источников. Мы надеемся, что читатель, не знакомый с этими формализмами, сможет получить достаточное представление об их базовых идеях на основе примеров, которые приводятся ниже.

Подобно языкам программирования, синтаксис λ -термов может быть задан при помощи БНФ (форм Бэкуса-Наура):

$$Exp = Var \mid Const \mid Exp\ Exp \mid \lambda\ Var.\ Exp$$

после чего мы можем трактовать их, как это принято в теории формальных языков, не просто как цепочки символов, а как абстрактные синтаксические деревья. Это значит, что соглашения наподобие левоассоциативности операции применения функции либо интерпретации $\lambda x\ y.\ s$ как $\lambda x.\ \lambda y.\ s$, а также неразличимость переменных и констант по именам не являются неотъемлемой частью формализма λ -исчисления, а имеют смысл исключительно в момент преобразования терма в форму, подходящую для восприятия человеком, либо в обратном направлении.

В завершение упомянем ещё одно соглашение, принятое в данном пособии. Будем использовать в λ -термах односимвольные имена не только для констант и переменных, но и для так называемых *метапеременных*, обозначающих любые термы. Например, выражение $\lambda x.\ s$ может представлять как константную функцию со значением s , так и произвольную λ -абстракцию по переменной x . Для предотвращения путаницы, условимся применять буквы s , t и u в качестве метапеременных. Устранить неоднозначность полностью возможно, например, за счёт потери компактности записи: введением имён переменных вида V_x вместо x , а констант — C_k вместо k , после чего исчезает необходимость приписывать именам особый статус.

2.3.2 Свободные и связанные переменные

В этом разделе мы формализуем интуитивное понятие свободных и связанных переменных, которое, между прочим, служит хорошим примером определения примитивно-рекурсивных функций. Интуитивно, вхождение переменной в заданный терм считается свободным, если оно не лежит в области действия соответствующей абстракции. Обозначим множество свободных переменных в терме s через $FV(s)$ и дадим его рекурсивное определение:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(c) &= \emptyset \\ FV(st) &= FV(s) \cup FV(t) \\ FV(\lambda x.\ s) &= FV(s) - \{x\} \end{aligned}$$

Аналогично вводится и понятие множества связанных переменных $BV(s)$:

$$BV(x) = \emptyset$$

$$\begin{aligned}
BV(c) &= \emptyset \\
BV(st) &= BV(s) \cup BV(t) \\
BV(\lambda x. s) &= BV(s) \cup \{x\}
\end{aligned}$$

Например, если $s = (\lambda x y. x) (\lambda x. z x)$, то $FV(s) = \{z\}$ и $BV(s) = \{x, y\}$. Отметим, что в общем случае переменная может быть одновременно и свободной, и связанной в одном и том же терме, как это было показано выше. Воспользуемся структурной индукцией, чтобы продемонстрировать доказательство утверждений о свойствах λ -термов на примере следующей теоремы (аналогичные рассуждения применимы и ко множеству BV).

Теорема 2.1 *Для произвольного λ -терма s множество $FV(s)$ конечно.*

Доказательство: Применим структурную индукцию. Очевидно, что для терма s , имеющего вид переменной либо константы, множество $FV(s)$ конечно по определению (содержит единственный элемент либо пусто соответственно). Если терм s представляет собой комбинацию t и u , то согласно индуктивному предположению, как $FV(t)$, так и $FV(u)$ конечны, в силу чего $FV(s) = FV(t) \cup FV(u)$ также конечно, как объединение двух конечных множеств. Наконец, если s имеет форму $\lambda x. t$, то по определению $FV(s) = FV(t) - \{x\}$, а $FV(t)$ конечно по индуктивному предположению, откуда следует, что $FV(s)$ также конечно, поскольку его мощность не может превышать мощности $FV(t)$. \square

2.3.3 Подстановка

Одним из правил, которые мы хотим формализовать, является соглашение о том, что λ -абстракция и применение функции представляют собой взаимно обратные операции. То есть, если мы возьмём терм $\lambda x. s$ и применим его как функцию к терму-аргументу t , результатом будет терм s , в котором все свободные вхождения переменной x заменены термом t . Для большей наглядности это действие принято обозначать $\lambda x. s[x]$ и $s[t]$ соответственно.

Однако, простое на первый взгляд понятие подстановки одного терма вместо переменной в другой терм на самом деле оказалось весьма коварным, так что даже некоторые выдающиеся логики не избежали ложных утверждений относительно его свойств. Подобный грустный опыт разочаровывает довольно сильно, ведь как мы говорили ранее, одним из привлекательных свойств формальных правил служит возможность их чисто механического применения.

Обозначим операцию подстановки терма s вместо переменной x в другой терм t как $t[s/x]$. Иногда можно встретить другие обозначения, например, $t[x:=s]$, $[s/x]t$, или даже $t[x/s]$. Мы полагаем, что предложенный нами вариант легче всего запомнить по аналогии с умножением дробей: $x[t/x] = t$. На первый взгляд, рекуррентное определение понятия подстановки выглядит так:

$$\begin{aligned}
x[t/x] &= t \\
y[t/x] &= y, \text{ если } x \neq y \\
c[t/x] &= c \\
(s_1 s_2)[t/x] &= s_1[t/x] s_2[t/x] \\
(\lambda x. s)[t/x] &= \lambda x. s \\
(\lambda y. s)[t/x] &= \lambda y. (s[t/x]), \text{ если } x \neq y
\end{aligned}$$

К сожалению, это определение не совсем верно. Например, подстановка $(\lambda y. x + y)[y/x] = \lambda y. y + y$ не соответствует интуитивным ожиданиям от ее результата.³ Исходный λ -терм интерпретировался как функция, прибавляющая x к своему аргументу, так что после подстановки мы ожидали получить функцию, которая прибавляет y , а на деле получили функцию, которая свой аргумент удваивает. Источником проблемы служит *захват* переменной y , которую мы подставляем, операцией $\lambda y. \dots$, которая связывает одноименную переменную. Чтобы этого не произошло, связанную переменную требуется предварительно переименовать:

$$(\lambda y. x + y) = (\lambda w. x + w),$$

а лишь затем производить подстановку:

$$(\lambda w. x + w)[y/x] = \lambda w. y + w$$

Существуют два подхода к решению этой проблемы. С одной стороны, можно условиться, что подстановка недопустима в ситуации захвата переменной, а с другой — расширить формальное определение подстановки таким образом, чтобы требуемое переименование переменных происходило автоматически. Мы остановимся на последнем варианте:

$$\begin{aligned} x[t/x] &= t \\ y[t/x] &= y, \text{ если } x \neq y \\ c[t/x] &= c \\ (s_1 s_2)[t/x] &= s_1[t/x] s_2[t/x] \\ (\lambda x. s)[t/x] &= \lambda x. s \\ (\lambda y. s)[t/x] &= \lambda y. (s[t/x]), \text{ если } x \neq y \text{ и либо } x \notin FV(s), \text{ либо } y \notin FV(t) \\ (\lambda y. s)[t/x] &= \lambda z. (s[z/y][t/x]) \text{ в противном случае, причём } z \notin FV(s) \cup FV(t) \end{aligned}$$

Единственное отличие этого определения заключается в двух последних правилах. Мы следуем предыдущему определению в двух безопасных ситуациях, когда либо переменная x не свободна в терме s , так что подстановка оказывается тривиальной, либо когда y не свободна в t , так что захват переменной не произойдет (на данном уровне). Однако, в случае, когда оба эти условия не выполняются, мы предварительно переименовываем переменную y в z , выбранную так, чтобы она не была свободной ни в терме s , ни в терме t , после чего продолжаем, как описано выше. Для определенности, переменная z может выбираться некоторым фиксированным способом, например, как первая в лексикографическом порядке имен переменная из множества всех переменных, не имеющих свободных вхождений ни в s , ни в t .⁴

³Строго говоря, нам следовало бы писать $+ x y$, нежели $x + y$, но мы будем, как прежде, использовать стандартные операции в инфиксной форме.

⁴Знатоки могут быть обеспокоены тем, что последнее правило не позволяет считать это определение *примитивно*-рекурсивным. Однако, его легко преобразовать в примитивно-рекурсивную множественную параллельную подстановку. Подобная процедура напоминает усиление индуктивного предположения в ходе доказательства. Отметим, что по построению пара подстановок в последнем правиле может осуществляться как параллельно, так и последовательно без ущерба для результата.

2.3.4 Преобразования

Ещё одной из основ λ -исчисления служат три «преобразования» — операции получения по заданному терму другого, равного ему в интуитивном смысле. Традиционно их обозначают буквами греческого алфавита: α (альфа), β (бета) и η (эта).⁵ Приведём формальные определения этих операций, обозначив каждую из них помеченной стрелкой.

- Альфа-преобразование: $\lambda x. s \xrightarrow{\alpha} \lambda y. s[y/x]$, при условии, что $y \notin FV(s)$. Например, $\lambda u. u \ v \xrightarrow{\alpha} \lambda w. w \ v$, но $\lambda u. u \ v \not\xrightarrow{\alpha} \lambda v. v \ v$. Такое ограничение устраняет возможность еще одного случая захвата переменной.
- Бета-преобразование: $(\lambda x. s) t \xrightarrow{\beta} s[t/x]$.
- Эта-преобразование: $\lambda x. t \ x \xrightarrow{\eta} t$, если $x \notin FV(t)$. Например, $\lambda u. v \ u \xrightarrow{\eta} v$, но $\lambda u. u \ u \not\xrightarrow{\eta} u$.

Среди этих трех операций наиболее важной для нас является β -преобразование, поскольку оно соответствует вычислению функции для заданного аргумента. В то же время, α -преобразование играет роль вспомогательного средства переименования связанных переменных, а η -преобразование представляет собой разновидность *экстенциональности*, в силу чего интересно главным образом с точки зрения логика, а не программиста.

2.3.5 Равенство λ -выражений

Используя приведённые выше правила преобразований, мы можем определить формально условия, при которых два λ -терма считаются равными. В общих чертах, два терма равны, если один из них может быть получен из другого в ходе конечной последовательности α , β либо η -преобразований, которые применяются к произвольным подтермам как в прямом, так и в обратном направлении. Другими словами, отношение λ -равенства представляет собой *congruence closure* трёх преобразований и обладает свойствами рефлексивности, симметричности, транзитивности и заменяемости. Ниже приводится формальное индуктивное определение, правила которого трактуются следующим образом: если утверждение над горизонтальной чертой выполняется, то справедливо и утверждение под ней.

$$\begin{array}{c}
 \frac{s \xrightarrow{\alpha} t \text{ или } s \xrightarrow{\beta} t \text{ или } s \xrightarrow{\eta} t}{s = t} \\
 \\
 \frac{}{t = t} \\
 \frac{}{s = t} \\
 \frac{}{t = s} \\
 \frac{s = t \text{ и } t = u}{s = u}
 \end{array}$$

⁵Такие имена преобразований были введены Карри. Первоначально Чёрч называл α - и β -преобразования «правило действий I» и «правило действий II» соответственно.

$$\frac{s = t}{s \ u = t \ u}$$

$$\frac{s = t}{u \ s = u \ t}$$

$$\frac{s = t}{\lambda x. s = \lambda x. t}$$

Отметим, что использование обычного знака равенства ($=$) в данном контексте может ввести в заблуждение. В самом деле, мы *задаём* некоторое отношение λ -равенства, взаимосвязь которого с понятием равенства соответствующих математических объектов остаётся неясной.⁶ В то же время очевидно, что следует отличать λ -равенство от *синтаксического*. Последнее будем называть «тождеством» и обозначим символом \equiv . Например, $\lambda x. x \not\equiv \lambda y. y$, хотя в то же время $\lambda x. x = \lambda y. y$.

Во многих случаях оказывается, что α -преобразования не играют роли, в силу чего вместо строгого тождества применяется его вариант \equiv_α . Это отношение определяется подобно λ -равенству, но исключительно для α -преобразований. Например, $(\lambda x. x)y \equiv_\alpha (\lambda y. y)y$. Многие авторы используют его как тождество λ -термов, тем самым разбивая множество термов на соответствующие классы эквивалентности. Существуют альтернативные системы обозначений, например (de Bruijn 1972), в которых связанные переменные не имеют имён. В таких системах традиционное понятие тождества совпадает с \equiv_α .

2.3.6 Экстенциональность

Мы уже упоминали ранее, что η -преобразование воплощает принцип *экстенциональности*. В рамках общепринятых философских понятий два свойства называются *экстенционально эквивалентными* (либо *коэкстенсивными*), если этими свойствами обладают в точности одни и те же объекты. В теории множеств принята аксиома экстенциональности, согласно которой два множества совпадают, если они состоят из одних и тех же элементов. Аналогично, будем говорить, что две функции эквивалентны, если области их определения совпадают, а значения функций для всевозможных аргументов также одинаковы.

Введение η -преобразования делает наше понятие λ -равенства экстенциональным. В самом деле, пусть $f \ x$ и $g \ x$ равны для произвольного значения x ; в частности, $f \ y = g \ y$, где переменная y выбирается так, чтобы она не была свободной как в f , так и в g . Согласно последнему из приведённых выше правил эквивалентности, $\lambda y. f \ y = \lambda y. g \ y$. Применив дважды η -преобразование, получаем, что $f = g$. С другой стороны, из экстенциональности следует, что всевозможные η -преобразования не нарушают эквивалентности, поскольку согласно правилу β -редукции $(\lambda x. t \ x) \ y = t \ y$ для произвольного y , если переменная x не является свободной в терме t . На этом мы завершаем обсуждение сущности η -преобразования и его влияния на теорию в целом, чтобы уделить больше внимания более перспективному с точки зрения вычислимости β -преобразованию.

⁶ Действительно, ведь мы не определяем достаточно точно, каково это соответствие *само по себе*. Тем не менее, существуют модели λ -исчисления, в которых λ -равенство трактуется как обычное.

2.3.7 λ -редукция

Отношение λ -равенства, как и следовало ожидать, является симметричным. Оно достаточно хорошо соответствует интуитивному понятию эквивалентности λ -термов, но с алгоритмической точки зрения более интересен его несимметричный аналог. Определим отношение *редукции* \longrightarrow следующим образом:

$$\begin{array}{c}
 \frac{s \xrightarrow{\alpha} t \text{ или } s \xrightarrow{\beta} t \text{ или } s \xrightarrow{\eta} t}{s \longrightarrow t} \\
 \\
 \frac{}{t \longrightarrow t} \\
 \\
 \frac{s \longrightarrow t \text{ и } t \longrightarrow u}{s \longrightarrow u} \\
 \\
 \frac{s \longrightarrow t}{s u \longrightarrow t u} \\
 \\
 \frac{s \longrightarrow t}{u s \longrightarrow u t} \\
 \\
 \frac{s \longrightarrow t}{\lambda x. s \longrightarrow \lambda x. t}
 \end{array}$$

В действительности слово «редукция» (в частности, термин β -редукция, которым иногда называют β -преобразования) не отражает точно сути происходящего, поскольку в процессе редукции терм может увеличиваться, например:

$$\begin{aligned}
 (\lambda x. x x x) (\lambda x. x x x) &\longrightarrow (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) \\
 &\longrightarrow (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) \\
 &\longrightarrow \dots
 \end{aligned}$$

Однако, несмотря на это редукция имеет прямое отношение к процедуре вычисления терма, в ходе которой последовательно вычисляются комбинации вида $f(x)$, где f — λ -абстракция. Если на некотором этапе оказывается, что не могут быть применены никакие правила редукции, кроме α -преобразований, то говорят, что терм имеет *нормальную форму*.

2.3.8 Стратегии редукции

Отложив на время наши теоретические рассуждения, напомним их взаимосвязь с практикой функционального программирования. Программа на функциональном языке представляет собой *выражение*, а её выполнение — вычисление этого выражения. То есть, в терминах, изложенных выше, мы собираемся начать процесс вычислений с соответствующего терма и применять к нему правила редукции до тех пор, пока это возможно. Возникает вопрос: какое из имеющихся правил следует применять на каждом этапе? Отношение редукции — недетерминированное, то есть, для некоторых термов t найдётся множество термов t_i таких, что $t \longrightarrow t_i$. Выбор того или иного варианта оказывается иногда принципиально важным, поскольку может привести как к конечной, так и к бесконечной последовательности редукций (выполнение соответствующей программы при этом либо завершается,

либо заикливается). Например, подвергая редукции наиболее глубокий *редекс*⁷ в выражении, приведённом ниже, мы получаем бесконечную последовательность редукций:

$$\begin{aligned}
 & (\lambda x. y) ((\lambda x. x x x) (\lambda x. x x x)) \\
 \longrightarrow & (\lambda x. y) ((\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x)) \\
 \longrightarrow & (\lambda x. y) ((\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x)) \\
 \longrightarrow & \dots
 \end{aligned}$$

В то же время, редукция самого внешнего редекса

$$(\lambda x. y) ((\lambda x. x x x) (\lambda x. x x x)) \longrightarrow y$$

немедленно ведёт нас к желаемому результату.

Значение выбора стратегии редукции окончательно проясняется следующими теоремами, которые мы рассмотрим без доказательств, поскольку они слишком велики для данного учебника. Первая теорема утверждает, что ситуация, с которой мы столкнулись в последнем примере, и её решение достаточно общие, т. е. стратегия редукции самого левого редекса является наилучшей с точки зрения её завершимости.

Теорема 2.2 *Если справедливо $s \longrightarrow t$, где терм t имеет нормальную форму, то последовательность редукций, которая начинается с терма s и состоит в применении правил редукции к самому левому редексу, всегда завершается и приводит к терму в нормальной форме.*

Применение этой теоремы требует формального определения понятия *самого левого редекса*: для терма $(\lambda x. s) t$ это он сам, для произвольного другого терма вида $s t$ самым левым является самый левый редекс s , наконец, для абстракции $\lambda x. s$ это тоже самый левый редекс s . В рамках принятых в данном пособии обозначений мы будем всегда выбирать такой редекс, чтобы соответствующий ему символ λ был расположен левее прочих.

2.3.9 Теорема Чёрча-Россера

Следующее утверждение, которое мы рассмотрим, широко известно как теорема Чёрча-Россера. Оно гласит, что для двух конечных последовательностей редукций, начатых с терма t , всегда найдутся две другие последовательности, сводящие результаты предыдущих к одному и тому же терму (который, впрочем, может и не быть в нормальной форме).

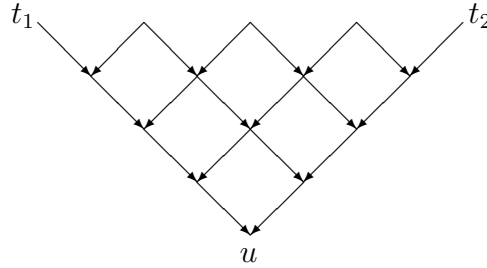
Теорема 2.3 *Если $t \longrightarrow s_1$ и $t \longrightarrow s_2$, то существует терм u такой, что $s_1 \longrightarrow u$ и $s_2 \longrightarrow u$.*

Важные следствия данного утверждения:

⁷англ. *redex* (reducible expression) — «редуцируемое выражение»

Corollary 2.4 Если $t_1 = t_2$ то найдётся терм u такой, что $t_1 \longrightarrow u$ и $t_2 \longrightarrow u$.

Доказательство: Легко показать (при помощи структурной индукции), что отношение λ -равенства = представляет собой симметричное транзитивное замыкание отношения редукции. Дальнейшее следует по индукции согласно свойствам симметричного транзитивного замыкания. Приведённая ниже диаграмма может показаться читателям, не склонным к формальным построениям, более доходчивой:



Мы полагаем, что $t_1 = t_2$, т. е. существует некоторая последовательность редукций в обеих направлениях (зигзагообразная линия в верхней части рисунка), которая их объединяет. Теорема Чёрча-Россера позволяет нам заполнить недостающие участки на краях диаграммы, после чего требуемый результат достигается композицией этих редукций.

Corollary 2.5 Если $t = t_1$ и $t = t_2$, причём t_1 и t_2 имеют нормальную форму, то $t_1 \equiv_\alpha t_2$, т. е. t_1 и t_2 равны с точностью до α -преобразований.

Доказательство: Согласно изложенному выше, найдётся некоторый терм u такой, что $t_1 \longrightarrow u$ и $t_2 \longrightarrow u$. Но так как t_1 и t_2 уже имеют нормальную форму, последовательность редукций, приводящая к терму u , может состоять лишь из α -преобразований. \square

Таким образом, нормальные формы, если они существуют, являются единственными с точностью до α -преобразования. Это даёт нам первое обоснование того, что отношение λ -равенства нетривиально, т. е. что существуют неравные термы. Например, поскольку $\lambda x y. x$ и $\lambda x y. y$ несводимы друг к другу исключительно при помощи α -преобразований, они не равны.

Подытожим важность полученных результатов в свете теории вычислимости. Стратегия редукции, в которой на каждом шаге выбирается самый левый редекс, считается, в известном смысле, наилучшей, поскольку она применима всегда, когда применима любая другая стратегия. Такая стратегия получила название *нормального порядка редукции*. С другой стороны, *любая* другая конечная последовательность редукций будет всегда давать тот же самый результат; более того, всегда остаётся возможность прекратить применение этой стратегии, перейдя при необходимости к нормальному порядку. Мы увидим практическое применение этого принципа далее.

2.4 Комбинаторы

Впервые понятие комбинатора и основанная на нём теория были сформулированы М.И. Шейнфинкелем в работе [Schönfinkel \(1924\)](#) ещё до появления λ -исчисления. Вскоре после этого аналогичные результаты были получены Карри, независимо от Шейнфинкеля и Чёрча. (Когда Карри ознакомился с работами Шейнфинкеля, он предпринял попытку с ним связаться, но к этому времени Шейнфинкель оказался в психиатрической лечебнице.) В данной работе мы позволим себе не соблюдать историческую достоверность, изложив теорию комбинаторов как один из аспектов λ -исчисления.

Будем называть *комбинатором* терм λ -исчисления без свободных переменных. Такие термы также принято называть *замкнутыми*, поскольку их значение не зависит от значений каких-либо переменных. В дальнейшем в курсе функционального программирования мы встретимся с большим количеством полезных комбинаторов, но краеугольным камнем теории комбинаторов служит тот факт, что на самом деле достаточно лишь лишь немногих из них. Оказывается, что произвольный терм может быть выражен при помощи определённого множества комбинаторов и всевозможных переменных, операция λ -абстракции становится ненужной. В частности, замкнутый терм может быть представлен исключительно через эти комбинаторы. Дадим их определения:

$$\begin{aligned} I &= \lambda x. x \\ K &= \lambda x y. x \\ S &= \lambda f g x. (f x)(g x) \end{aligned}$$

Чтобы легче их запомнить, можно воспользоваться простыми мнемоническими правилами.⁸ Комбинатор I представляет собой тождественную функцию («идентичность»), комбинатор K порождает семейство константных⁹ функций: после применения к аргументу a он даёт функцию $\lambda y. a$. Наконец, S — комбинатор «совместного применения», который принимает в качестве аргументов две функции, применяемые к общему аргументу. Докажем следующее утверждение:

Лемма 2.6 *Для произвольного λ -терма t , не содержащего λ -абстракций, найдётся терм u , который также не содержит λ -абстракций и представляет собой композицию S , K , I и переменных, причём $FV(u) = FV(t) - \{x\}$ и $u = \lambda x. t$, т. е. терм u λ -равен $\lambda x. t$.*

Доказательство: *Применим к терму t структурную индукцию. Согласно условию, он не может быть абстракцией, поэтому нам требуется рассмотреть лишь три случая.*

- Если t представляет собой переменную, возможны два случая, из которых непосредственно следует требуемый вывод: при $t = x$ мы получаем $\lambda x. x = I$, иначе, например, при $t = y$, $\lambda x. y = K y$.

⁸Мы не утверждаем, что эти правила имеют под собой историческую основу.

⁹Шейнфинкель использовал обозначение C , но мы предлагаем своё, основанное на нем. Konstant, в честь его немецкого происхождения (автор ошибается, М. И. Шейнфинкель работал в Германии, но родился в России — Прим. перев.)

- Если t — константа c , то $\lambda x. c = K c$.
- Если t представляет собой комбинацию термов, например, s и u , то согласно индуктивному предположению найдутся термы s' и u' , которые не содержат λ -абстракций и для которых справедливы равенства $s' = \lambda x. s$ и $u' = \lambda x. u$. Из этого можно сделать вывод, что $S s' u'$ является искомым выражением. В самом деле,

$$\begin{aligned}
 S s' u' x &= S (\lambda x. s) (\lambda x. u) x \\
 &= ((\lambda x. s) x)((\lambda x. u) x) \\
 &= s u \\
 &= t
 \end{aligned}$$

Таким образом, применив η -преобразование, мы получаем $S s' u' = \lambda x. S s' u' x = \lambda x. t$, поскольку согласно индуктивному предположению переменная x не является свободной в термах s' либо u' .

□

Теорема 2.7 Для произвольного λ -терма t существует не содержащий λ -абстракций терм t' , полученный композицией K , I и переменных, такой, что $FV(t') = FV(t)$ и $t' = t$.

Доказательство: Применим структурную индукцию к терму t и воспользуемся леммой 2.6. Например, если терм t имеет вид $\lambda x. s$, то мы сначала можем получить, согласно индуктивному предположению, терм s' — свободный от λ -абстракций эквивалент s . Далее применим лемму к $\lambda x. s'$. Прочие случаи очевидны.

□

Это примечательное утверждение может быть даже усилено, поскольку комбинатор I выражается через S и K . Отметим, что для произвольного A

$$\begin{aligned}
 S K A x &= (K x)(A x) \\
 &= (\lambda y. x)(A x) \\
 &= x
 \end{aligned}$$

Отсюда, применив η -преобразование, получаем, что $I = S K A$ для любых A . Однако, по причинам, которые станут яснее после знакомства с понятием типа, наиболее удобно положить $A = K$. Таким образом, $I = S K K$, что даёт нам возможность устранить все вхождения I в комбинаторные выражения.

Заметим, что приведённые выше доказательства имеют конструктивный характер, поскольку предлагают конкретные процедуры получения по заданному терму эквивалентного комбинаторного выражения. Процесс его построения идёт в направлении снизу вверх, и для каждой λ -абстракции, которая по построению имеет тело, свободное от λ -абстракций, применяются сверху вниз преобразования, изложенные в лемме.

Несмотря на то, что мы рассматриваем комбинаторы как некоторые термы λ -исчисления, на их основе можно сформулировать независимую теорию. Её построение начинается с определения формальных правил конструирования

выражений, в которые не входит λ -абстракция, но входят комбинаторы. Далее вместо α , β и η -преобразований вводятся правила преобразования для выражений, включающих комбинаторы, например, $K\ x\ y \longrightarrow x$. Такая теория будет иметь множество аналогий в традиционном λ -исчислении, например, теорема Чёрча-Россера оказывается справедливой и для приведённого выше определения редукции. Кроме того, полностью устраняются сложности со связыванием переменных. Тем не менее, мы считаем полученный формализм не слишком интуитивным, поскольку комбинаторные выражения нередко бывают весьма неясными.

Помимо важной роли, которую комбинаторы играют в логике, они также имеют и определённый практический потенциал. Как мы уже кратко упоминали (подробное изложение ожидается в следующих разделах), λ -исчисление может считаться простым функциональным языком, основой для более развитых и практически применимых языков, наподобие ML. Теорема о комбинаторной полноте даёт основание говорить, что выражения λ -исчисления могут быть «скомпилированы» в «машинный код» комбинаторов. Эта терминология из теории языковых процессоров оказывается на самом деле вполне уместной. Комбинаторы применялись как средство реализации функциональных языков, в том числе и на уровне аппаратного обеспечения, предназначенного для вычисления комбинаторных выражений.

Дополнительная литература

Энциклопедическая, но при этом доступная работа по теории λ -исчисления — [Barendregt \(1984\)](#). Другой популярный учебник — [Hindley and Seldin \(1986\)](#). Обе эти книги содержат доказательства результатов, которые мы приводим без обоснования. Вторая часть [Gordon \(1988\)](#) представляет собой упрощённое изложение предмета, ориентированное на его применение в прикладной математике. Существенная часть данного курса базируется на последней работе.

Упражнения

1. Найдите нормальную форму терма $(\lambda x\ x\ x.\ x)\ a\ b\ c$.
2. Пусть $twice = \lambda f\ x.\ f(fx)$. Каков интуитивный смысл $twice$? Найдите нормальную форму $twice\ twice\ twice\ f\ x$. (Напомним, что операция применения функции левоассоциативна.)
3. Найдите терм t такой, что $t \xrightarrow{\beta} t$. Можно ли утверждать, что терм имеет нормальную форму тогда и только тогда, когда из $t \longrightarrow t'$ следует $t \equiv_{\alpha} t'$?
4. В каком случае справедливо $s[t/x][u/y] \equiv_{\alpha} s[u/y][t/x]$?
5. Постройте выражение, равносильное $\lambda f\ x.\ f(x\ x)$, используя лишь комбинаторы I , K и S .
6. Найдите *единственный* комбинатор X такой, что все λ -термы эквивалентны термам, построенным композицией X и переменных. Указание: положите $A = \lambda p.\ p\ K\ S\ K$, а затем рассмотрите $A\ A\ A$ и $A\ (A\ A)$.

7. Докажите, что терм X является комбинатором неподвижной точки тогда и только тогда, когда он представляет собой неподвижную точку комбинатора G такого, что $G = \lambda y\ m.\ m(y\ m)$.

Глава 3

λ -исчисление как язык программирования

«Проблема разрешимости» (также известная как *Entscheidungsproblem*) была одним из основных предметов изучения логиков 1930-х. Формулировка задачи такова: существует ли некоторая систематическая (механическая) процедура определения истинности утверждения в логике первого порядка? Положительный ответ на этот вопрос имел бы фундаментальное философское и, возможно, практическое значение: принципиальную возможность решить большое количество разнообразных сложных математических задач исключительно при помощи некоторого фиксированного метода (в настоящий момент используется термин *алгоритм*) без привлечения дополнительных творческих усилий.

Очевидно, что проблема разрешимости не проста, поскольку требует точного определения понятия «систематической» либо «механической» процедуры на языке математики. В работе [Turing \(1936\)](#) даётся, вероятно, лучший анализ этой задачи. В ней также утверждается, что механическими можно считать действия, которые могут быть в принципе выполнены достаточно умным клерком, не обладающим знаниями об объекте этих действий. Абстрагирование поведения такого клерка привело в дальнейшем к известному понятию «машины Тьюринга». Вопреки тому, что эта концепция была чисто математической, а роль исполнителя действий первоначально предназначалась человеку, мы можем также рассматривать машину Тьюринга как очень простой компьютер. Несмотря на простоту, эта машина способна проделать любые вычисления, доступные реальным машинным исполнителям.¹ Вычислительную модель, эквивалентную по своей полноте машине Тьюринга, принято называть *полной по Тьюрингу* либо *Тьюринг-полной*.

Почти одновременно с Тьюрингом, другими авторами были предложены независимые определения понятия «механической процедуры», большая часть которых оказалась эквивалентной машине Тьюринга по своей вычислительной полноте. В частности, лямбда-исчисление, первоначально предназначенное на роль формальной основы математики, также возможно трактовать и как язык программирования, в котором исполнение программ сводится к последовательности

¹В действительности, машина Тьюринга обладает большей вычислительной полнотой за счет отсутствия ограничений на объем доступной памяти. Строго говоря, любой существующий компьютер алгоритмически эквивалентен конечному автомату, но для удобства принято также полагать объем его памяти бесконечным.

бета-преобразований. В самом деле, Чёрч ещё до публикации работ Тьюринга сделал предположение, что множество операций, представимых в рамках лямбда-исчисления, является формальным эквивалентом интуитивного понятия «механической процедуры». Этот постулат был в дальнейшем назван *тезисом Чёрча*. В работе Church (1936) показано, что из этого тезиса следует неразрешимость *Entscheidungsproblem*. Тьюринг впоследствии доказал, что множество функций, представимых в рамках лямбда-исчисления, в точности совпадает со множеством функций, вычислимых машиной Тьюринга. Этот результат послужил ещё одним доводом в пользу справедливости тезиса Чёрча.

С точки зрения современных программистов, программы для машины Тьюринга могут считаться достаточно примитивной разновидностью машинных кодов. В самом деле, очень вероятно, что именно машины Тьюринга, в особенности так называемая «универсальная машина»,² оказали решающее влияние на разработку современных компьютерных архитектур с хранимой программой; впрочем, степень этого влияния и его природа продолжают служить объектом дискуссий (Robinson 1994). Примечательно то, что некоторые другие альтернативные определения «механической процедуры», часто сформулированные задолго до появления электронных компьютеров, достаточно точно соответствуют реальным методам программирования. Например, алгоритмы Маркова (формальная вычислительная модель, популярная в Советском Союзе) могут считаться основой языка программирования SNOBOL. В дальнейшем нас будет интересовать аналогичное влияние лямбда-исчисления на эволюцию функциональных языков.

Язык LISP, второй (после FORTRAN) старейший язык высокого уровня, использует некоторые понятия лямбда-исчисления, в частности, обозначение (LAMBDA ...) для безымянных функций, но в целом ему не соответствует. В самом деле, как ранние версии языка, так и некоторые его современные диалекты используют принцип *динамического связывания* имён переменных, несовместимый с лямбда-исчислением (подробное обсуждение см. ниже). Более того, в ранних версиях отсутствовала приемлемая поддержка понятия функций высших порядков, зато имелся существенный объём императивных конструкций. Но несмотря на это, LISP заслуживает внимания как первый функциональный язык программирования, в котором также впервые были реализованы многие сопутствующие возможности, такие как автоматическое распределение памяти и «сборка мусора».

Влияние лямбда-исчисления на языки программирования приобрело реальный вес с появлением в 1960-х работ Лэндина и Стрейчи. В частности, Лэндин показал, что множество свойств распространённых в то время (императивных) языков может успешно анализироваться в терминах лямбда-исчисления, к примеру, понятие областей видимости переменных в Algol 60. В работе Landin (1966) было предложено использовать лямбда-исчисление как основу языков программирования, примером чего послужил функциональный язык ISWIM («If you See What I Mean» — досл. «Если вам понятно, о чём речь»). Эта публикация завоевала в дальнейшем широкую известность и стала отправной точкой в разработке многих других языков, нашедших практическое применение.

Язык ML ведёт свою историю с появления в роли метаязыка (откуда,

²Универсальной называется машина Тьюринга, способная воспроизводить работу любой другой, т. е. выполнять роль *интерпретатора*. Подробнее это будет рассмотрено в курсе Computation Theory.

собственно, и происходит его название ML, Meta Language) системы доказательства теорем Edinburgh LCF (Gordon, Milner, and Wadsworth 1979). Это значит, что язык был предназначен для реализации алгоритмов логического вывода в формальном дедуктивном исчислении. Определение языка обнаруживает существенное влияние ISWIM, но в отличие от последнего, ML был расширен такими возможностями, как новаторская полиморфная типизация, включающая абстрактные типы данных, либо система обработки ошибок на основе *исключений*. Эти черты языка были введены, исходя из реальных практических потребностей, что в итоге привело к целостному и точному дизайну. Подобная узкая специализация характерна для успешных языков (язык C может служить ещё одним хорошим примером) и резко их отличает от провальных попыток коллективного проектирования, таких как Algol 60, который оказался скорее источником важных идей, нежели практичным инструментом. Дальнейшее знакомство с ML состоится позже, а в данный момент мы рассмотрим, как в роли языка программирования может быть использовано *чистое лямбда-исчисление*.

3.1 Представление данных в λ -исчислении

Программы для своей работы требуют входных данных, поэтому мы начнём с фиксации определенного способа представления данных в виде выражений лямбда-исчисления. Далее введём некоторые базовые операции над этим представлением. Во многих случаях оказывается, что выражение s , представленное в форме, удобной для восприятия человеком, может напрямую отображаться в лямбда-выражение s' . Этот процесс получил жаргонное название «синтаксическая глазировка» («syntactic sugaring»), поскольку делает горькую пилюлю чистой лямбда-нотации более удобоваримой. Введём следующее обозначение:

$$s \triangleq s'.$$

Будем говорить, что « $s = s'$ по определению»; другая общепринятая форма записи этого отношения — $s =_{def} s'$. При желании, мы можем всегда считать, что вводим некоторое константное выражение, определяющее семантику операции, которая затем применяется к своим аргументам в обычном стиле лямбда-исчисления, абстрагируясь тем самым от конкретных обозначений. Например, выражение `if E then E_1 else E_2` возможно трактовать как `COND E E_1 E_2` , где `COND` — некоторая константа. В подобном случае все переменные в левой части определения должны быть связаны операцией абстракции, т. е. вместо

$$\text{fst } p \triangleq p \text{ true}$$

(см. ниже) мы можем написать

$$\text{fst} \triangleq \lambda p. p \text{ true}.$$

3.1.1 Логические значения и условия

Для представления логических значений `true` («истина») и `false` («ложь») годятся любые два различных лямбда-выражения, но наиболее удобно использовать

следующие:

$$\begin{aligned}\text{true} &\triangleq \lambda x y. x \\ \text{false} &\triangleq \lambda x y. y\end{aligned}$$

Используя эти определения, легко ввести понятие условного выражения, соответствующего конструкции `? : языка C`. Отметим, что это условное *выражение*, а не *оператор* (который не имеет смысла в данном контексте), поэтому наличие альтернативы обязательно.

$$\text{if } E \text{ then } E_1 \text{ else } E_2 \triangleq E E_1 E_2$$

В самом деле, мы имеем:

$$\begin{aligned}\text{if true then } E_1 \text{ else } E_2 &= \text{true } E_1 E_2 \\ &= (\lambda x y. x) E_1 E_2 \\ &= E_1\end{aligned}$$

и

$$\begin{aligned}\text{if false then } E_1 \text{ else } E_2 &= \text{false } E_1 E_2 \\ &= (\lambda x y. y) E_1 E_2 \\ &= E_2\end{aligned}$$

Определив условное выражение, на его базе легко построить весь традиционный набор логических операций:

$$\begin{aligned}\text{not } p &\triangleq \text{if } p \text{ then false else true} \\ p \text{ and } q &\triangleq \text{if } p \text{ then } q \text{ else false} \\ p \text{ or } q &\triangleq \text{if } p \text{ then true else } q\end{aligned}$$

3.1.2 Пары и кортежи

Определим представление упорядоченных пар следующим образом:

$$(E_1, E_2) \triangleq \lambda f. f E_1 E_2$$

Использование скобок не обязательно, хотя мы часто будем использовать их для удобства восприятия либо подчеркивания ассоциативности. На самом деле, мы можем трактовать запятую как инфиксную операцию наподобие `+`. Определив пару, как указано выше, зададим соответствующие операции извлечения компонент пары как:

$$\begin{aligned}\text{fst } p &\triangleq p \text{ true} \\ \text{snd } p &\triangleq p \text{ false}\end{aligned}$$

Легко убедиться, что эти определения работают, как требуется:

$$\begin{aligned}
 \text{fst } (p, q) &= (p, q) \text{ true} \\
 &= (\lambda f. f \ p \ q) \text{ true} \\
 &= \text{true } p \ q \\
 &= (\lambda x \ y. x) \ p \ q \\
 &= p
 \end{aligned}$$

и

$$\begin{aligned}
 \text{snd } (p, q) &= (p, q) \text{ false} \\
 &= (\lambda f. f \ p \ q) \text{ false} \\
 &= \text{false } p \ q \\
 &= (\lambda x \ y. y) \ p \ q \\
 &= q
 \end{aligned}$$

Построение троек, четвёрок, пятёрок и так далее вплоть до кортежей произвольной длины n производится композицией пар:

$$(E_1, E_2, \dots, E_n) = (E_1, (E_2, \dots, E_n))$$

Всё, что нам при этом потребуется — определение, что инфиксный оператор запятая правоассоциативен. Дальнейшее понятно без введения дополнительных соглашений. Например:

$$\begin{aligned}
 (p, q, r, s) &= (p, (q, (r, s))) \\
 &= \lambda f. f \ p \ (q, (r, s)) \\
 &= \lambda f. f \ p \ (\lambda f. f \ q \ (r, s)) \\
 &= \lambda f. f \ p \ (\lambda f. f \ q \ (\lambda f. f \ r \ s)) \\
 &= \lambda f. f \ p \ (\lambda g. g \ q \ (\lambda h. h \ r \ s))
 \end{aligned}$$

В последнем выражении для удобства восприятия было произведено альфа-преобразование. Несмотря на то, что кортежи представляют собой «плоскую» структуру данных, путём последовательной их композиции возможно представить произвольную конечную древовидную структуру. Наконец, если кто-то предпочитает традиционные функции, заданные на декартовом произведении, нашим каррированным функциям, преобразовать их друг в друга нетрудно:

$$\begin{aligned}
 \text{CURRY } f &\triangleq \lambda x \ y. f(x, y) \\
 \text{UNCURRY } g &\triangleq \lambda p. g \ (\text{fst } p) \ (\text{snd } p)
 \end{aligned}$$

Эти специальные операции над парами нетрудно обобщить на случай кортежей произвольной длины n . Например, мы можем задать функцию-селектор выборки i -го компонента из кортежа p . Обозначим эту операцию $(p)_i$, и определим её как $(p)_1 = \text{fst } p$, $(p)_i = \text{fst } (\text{snd}^{i-1} \ p)$. Аналогичным образом возможно обобщение CURRY и UNCURRY:

$$\begin{aligned}
 \text{CURRY}_n \ f &\triangleq \lambda x_1 \ \dots \ x_n. f(x_1, \dots, x_n) \\
 \text{UNCURRY}_n \ g &\triangleq \lambda p. g \ (p)_1 \ \dots \ (p)_n
 \end{aligned}$$

Воспользуемся обозначением $\lambda(x_1, \dots, x_n). t$ как сокращённой формой записи для

$$\text{UNCURRY}_n (\lambda x_1 \cdots x_n. t),$$

обеспечив тем самым естественную нотацию для функций над декартовыми произведениями.

3.1.3 Натуральные числа

Представим натуральное число n в виде³

$$n \triangleq \lambda f x. f^n x,$$

то есть, $0 = \lambda f x. x$, $1 = \lambda f x. f x$, $2 = \lambda f x. f (f x)$ и т. д. Такое представление получило название *нумералов Чёрча*, хотя его базовая идея была опубликована ранее в работе [Wittgenstein \(1922\)](#).⁴ Это представление не слишком эффективно, так как фактически представляет собой запись чисел в системе счисления по основанию 1: 1, 11, 111, 1111, 11111, 111111, С точки зрения эффективности можно разработать гораздо лучшие формы представления, к примеру, кортеж логических значений, который интерпретируется как двоичная запись числа. Впрочем, в данный момент нас интересует лишь принципиальная вычислимость, а нумералы Чёрча имеют различные удобные формальные свойства. Например, легко привести лямбда-выражения такой общеизвестной арифметической операции, как получение числа, следующего в натуральном ряду за данным, то есть, прибавление единицы к аргументу операции:

$$\text{SUC} \triangleq \lambda n f x. n f (f x)$$

В самом деле,

$$\begin{aligned} \text{SUC } n &= (\lambda n f x. n f (f x))(\lambda f x. f^n x) \\ &= \lambda f x. (\lambda f x. f^n x) f (f x) \\ &= \lambda f x. (\lambda x. f^n x)(f x) \\ &= \lambda f x. f^n (f x) \\ &= \lambda f x. f^{n+1} x \\ &= n + 1 \end{aligned}$$

Аналогично, легко реализуется проверка числа на равенство нулю:

$$\text{ISZERO } n \triangleq n (\lambda x. \text{false}) \text{true}$$

поскольку

$$\text{ISZERO } 0 = (\lambda f x. x)(\lambda x. \text{false}) \text{true} = \text{true}$$

и

$$\begin{aligned} \text{ISZERO } (n + 1) &= (\lambda f x. f^{n+1} x)(\lambda x. \text{false}) \text{true} \\ &= (\lambda x. \text{false})^{n+1} \text{true} \\ &= (\lambda x. \text{false})((\lambda x. \text{false})^n \text{true}) \\ &= \text{false} \end{aligned}$$

³Запись выражения $f^n x$ с параметром n применяется исключительно для удобства записи, а не в силу его цикличности.

⁴См. «6.021 A number is the exponent of an operation».

Сумма и произведение двух нумералов Чёрча:

$$\begin{aligned} m + n &\triangleq \lambda f x. m f (n f x) \\ m * n &\triangleq \lambda f x. m (n f) x \end{aligned}$$

В справедливости этих определений легко убедиться:

$$\begin{aligned} m + n &= \lambda f x. m f (n f x) \\ &= \lambda f x. (\lambda f x. f^m x) f (n f x) \\ &= \lambda f x. (\lambda x. f^m x) (n f x) \\ &= \lambda f x. f^m (n f x) \\ &= \lambda f x. f^m ((\lambda f x. f^n x) f x) \\ &= \lambda f x. f^m ((\lambda x. f^n x) x) \\ &= \lambda f x. f^m (f^n x) \\ &= \lambda f x. f^{m+n} x \end{aligned}$$

и

$$\begin{aligned} m * n &= \lambda f x. m (n f) x \\ &= \lambda f x. (\lambda f x. f^m x) (n f) x \\ &= \lambda f x. (\lambda x. (n f)^m x) x \\ &= \lambda f x. (n f)^m x \\ &= \lambda f x. ((\lambda f x. f^n x) f)^m x \\ &= \lambda f x. ((\lambda x. f^n x)^m x) \\ &= \lambda f x. (f^n)^m x \\ &= \lambda f x. f^{mn} x \end{aligned}$$

Несмотря на то, что эти операции на натуральных числах были определены достаточно легко, вычисление числа, предшествующего данному, гораздо сложнее. Нам требуется выражение PRE такое, что $PRE\ 0 = 0$ и $PRE\ (n + 1) = n$. Решение этой задачи было предложено Клини в работе [Kleene \(1935\)](#). Клини применил такой приём: для заданного $\lambda f x. f^n x$ требуется «отбросить» одно из применений f . В качестве первого шага введём на множестве пар функцию $PREFN$ такую, что

$$PREFN\ f\ (\text{true}, x) = (\text{false}, x)$$

и

$$PREFN\ f\ (\text{false}, x) = (\text{false}, f\ x)$$

Предположив, что подобная функция существует, можно показать, что $(PREFN\ f)^{n+1}(\text{true}, x) = (\text{false}, f^n x)$. В свою очередь, этого достаточно, чтобы задать функцию PRE , не испытывая особых затруднений. Определение $PREFN$, удовлетворяющее нашим нуждам, таково:

$$PREFN \triangleq \lambda f p. (\text{false}, \text{if fst } p \text{ then snd } p \text{ else } f(\text{snd } p))$$

В свою очередь,

$$PRE\ n \triangleq \lambda f x. \text{snd}(n\ (PREFN\ f)\ (\text{true}, x))$$

Доказательство корректности этого определения предлагается читателю в качестве упражнения.

3.2 Рекурсивные функции

Возможность определения рекурсивных функций является краеугольным камнем функционального программирования, поскольку в его рамках это единственный общий способ реализовать итерацию. На первый взгляд, сделать подобное средствами лямбда-исчисления невозможно. В самом деле, *именование* функций представляется неперменной частью рекурсивных определений, так как в противном случае неясно, как можно сослаться на функцию в её собственном определении, не заикливаясь. Тем не менее, существует решение и этой проблемы, которое, однако, удалось найти лишь ценой значительных усилий, подобно построению функции PRE.

Ключом к решению оказалось существование так называемых *комбинаторов неподвижной точки*. Замкнутый терм Y называется комбинатором неподвижной точки, если для произвольного терма f выполняется равенство $f(Y f) = Y f$. Другими словами, комбинатор неподвижной точки определяет по заданному терму f его фиксированную точку, т. е. находит такой терм x , что $f(x) = x$. Первый пример такого комбинатора, найденный Карри, принято обозначать Y . Своим появлением он обязан парадоксу Рассела, чем объясняется его другое популярное название — «парадоксальный комбинатор». Мы определили

$$R = \lambda x. \neg(x x),$$

после чего обнаружили справедливость

$$R R = \neg(R R)$$

Таким образом, $R R$ представляет собой неподвижную точку операции отрицания. Отсюда, чтобы построить универсальный комбинатор неподвижной точки, нам потребуется лишь обобщить данное выражение, заменив \neg произвольной функцией, заданной аргументом f . В результате мы получаем

$$Y \triangleq \lambda f. (\lambda x. f(x x))(\lambda x. f(x x))$$

Убедиться в справедливости этого определения несложно:

$$\begin{aligned} Y f &= (\lambda f. (\lambda x. f(x x))(\lambda x. f(x x))) f \\ &= (\lambda x. f(x x))(\lambda x. f(x x)) \\ &= f((\lambda x. f(x x))(\lambda x. f(x x))) \\ &= f(Y f) \end{aligned}$$

Однако, несмотря на математическую корректность, предложенное решение не слишком привлекательно с точки зрения программирования, поскольку оно справедливо лишь в смысле лямбда-эквивалентности, но не редукции (в последнем выражении мы применяли *обратное* бета-преобразование). С учётом этих соображений альтернативное определение Тьюринга может оказаться более предпочтительным:

$$T \triangleq (\lambda x y. y (x x y)) (\lambda x y. y (x x y))$$

(Доказательство справедливости $T f \longrightarrow f(T f)$ предоставляется читателю в качестве упражнения.) Однако, мы можем без особого ущерба для строгости

изложения считать, что Yf может подвергаться бета-редукции в соответствии с последовательностью редукции для рекурсивных функций. Рассмотрим, как комбинатор неподвижной точки (например, Y) может применяться для реализации рекурсии. Воспользуемся в качестве примера вычислением факториала. Мы хотим определить функцию `fact` следующим образом:

$$\text{fact}(n) = \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n)$$

Прежде всего, преобразуем эту функцию в эквивалентную:

$$\text{fact} = \lambda n. \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n)$$

которая, в свою очередь, эквивалентна

$$\text{fact} = (\lambda f n. \text{if ISZERO } n \text{ then } 1 \text{ else } n * f(\text{PRE } n)) \text{ fact}$$

Отсюда следует, что `fact` представляет собой неподвижную точку такой функции F :

$$F = \lambda f n. \text{if ISZERO } n \text{ then } 1 \text{ else } n * f(\text{PRE } n)$$

В результате всё, что нам потребуется, это положить $\text{fact} = Y F$. Аналогичным способом можно воспользоваться и в случае взаимно рекурсивных функций, т. е. множества функций, определения которых зависят друг от друга. Такие определения, как

$$\begin{aligned} f_1 &= F_1 f_1 \cdots f_n \\ f_2 &= F_2 f_1 \cdots f_n \\ \dots &= \dots \\ f_n &= F_n f_1 \cdots f_n \end{aligned}$$

могут быть при помощи кортежей преобразованы в одно:

$$(f_1, f_2, \dots, f_n) = (F_1 f_1 \cdots f_n, F_2 f_1 \cdots f_n, \dots, F_n f_1 \cdots f_n)$$

Положив $t = (f_1, f_2, \dots, f_n)$, видим, что каждая из функций в правой части равенства может быть вычислена по заданному t применением соответствующей функции-селектора: $f_i = (t)_i$. Применив абстракцию по переменной t , получаем уравнение в канонической форме $t = F t$, решением которого является $t = Y F$, откуда в свою очередь находятся значения отдельных функций.

3.3 Let-выражения

Возможность использования безымянных функций была нами ранее преподнесена как одно из достоинств лямбда-исчисления. Более того, имена оказались необязательными даже при определении рекурсивных функций. Однако, зачастую всё же удобно иметь возможность давать выражениям имена с тем, чтобы избежать утомительного повторения больших термов. Простая форма такого именования может быть реализована как ещё один вид синтаксической глазури поверх чистого лямбда-исчисления:

$$\text{let } x = s \text{ in } t \triangleq (\lambda x. t) s$$

Простой пример применения этой конструкции работает, как и ожидается:

$$(\text{let } z = 2 + 3 \text{ in } z + z) = (\lambda z. z + z) (2 + 3) = (2 + 3) + (2 + 3)$$

Мы можем добиться как последовательного, так и параллельного связывания множества имён с выражениями. Первый случай реализуется простым многократным применением конструкции связывания, приведённой выше. Во втором случае введём возможность одновременного задания множества связываний, отделяемых друг от друга служебным словом **and**:

$$\text{let } x_1 = s_1 \text{ and } \dots \text{ and } x_n = s_n \text{ in } t$$

Будем рассматривать эту конструкцию как синтаксическую глазурь для

$$(\lambda(x_1, \dots, x_n). t) (s_1, \dots, s_n)$$

Продemonстрируем различия в семантике последовательного и параллельного связывания на примере:

$$\text{let } x = 1 \text{ in let } x = 2 \text{ in let } y = x \text{ in } x + y$$

и

$$\text{let } x = 1 \text{ in let } x = 2 \text{ and } y = x \text{ in } x + y$$

дают в результате 4 и 3 соответственно.

В дополнение к этому разрешим связывать выражения с именами, за которыми следует список параметров; такая форма конструкции **let** представляет собой ещё одну разновидность синтаксической глазури, позволяющую трактовать $f x_1 \dots x_n = t$ как $f = \lambda x_1 \dots x_n. t$. Наконец, помимо префиксной формы связывания $\text{let } x = s \text{ in } t$ введём постфиксную, которая в некоторых случаях оказывается удобнее для восприятия:

$$t \text{ where } x = s$$

Например, мы можем написать так: $y < y^2 \text{ where } y = 1 + x$.

Обычно конструкции **let** и **where** интерпретируются, как показано выше, без привлечения рекурсии. Например,

$$\text{let } x = x - 1 \text{ in } \dots$$

связывает x с уменьшенным на единицу значением, которое уже было связано с именем x в охватывающем контексте, а не пытается найти неподвижную точку выражения $x = x - 1$.⁵ В случае, когда нам требуется рекурсивная интерпретация, это может быть указано добавлением служебного слова **rec** в конструкции связывания (т. е. использованием **let rec** и **where rec** соответственно). Например,

$$\text{let rec fact}(n) = \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n)$$

Это выражение может считаться сокращённой формой записи $\text{let fact} = Y F$, где

$$F = \lambda f n. \text{if ISZERO } n \text{ then } 1 \text{ else } n * f(\text{PRE } n),$$

как было показано выше.

⁵Неподвижная точка этого выражения существует, но соответствующий лямбда-терм не имеет нормальной формы, т. е. в известном смысле не определен. Семантика незавершимых термов достаточно сложна и не будет нами рассматриваться. В действительности, основной вопрос заключается в наличии у терма не нормальной, а так называемой *головной нормальной формы*, что подробнее рассматривается в работах [Barendregt \(1984\)](#) и [Abramsky \(1990\)](#).

3.4 Достижение уровня полноценного языка программирования

На данный момент мы ввели достаточно обширный набор средств «синтаксической глазировки», реализующих удобочитаемый синтаксис поверх чистого лямбда-исчисления. Примечательно то, что этих средств достаточно для определения функции факториала в форме, очень близкой к языку ML. В связи с этим возникает вопрос, уместно ли считать лямбда-исчисление, расширенное предложенными обозначениями, практически пригодным языком программирования?

В конечном счёте, программа представляет собой единственное выражение. Однако, использование `let` для именования различных важных подвыражений, делает вполне естественной трактовку программы как множества *определений* различных вспомогательных функций, за которыми следует итоговое выражение, например:

```
let rec fact(n) = if ISZERO n then 1 else n * fact(PRE n) in
...
fact(6)
```

Эти определения вспомогательных функций могут трактоваться с математической точки зрения как уравнения. Подобная интерпретация не задаёт никаких ограничений ни на способ вычисления выражений, ни даже направление, в котором уравнения будут использоваться. Благодаря этому, функциональный подход к программированию часто называют *декларативным* наряду с логическим (примером последнего служит язык PROLOG).⁶ В рамках такого подхода программа не содержит явных инструкций, а лишь объявляет некоторые свойства соответствующих понятий, оставляя подробности своего выполнения компьютеру.

В то же время, программа бесполезна или, по крайней мере, неоднозначна, если для неё не определены некоторые содержательные действия компьютера. Следовательно, требуется понимание того, что внешне полностью декларативная программа должна быть выполнена неким определённым образом. В самом деле, вычисление выражения начинается с раскрытия всех входящих в него имён определений (т. е. уравнения интерпретируются слева направо), после чего производится последовательность β -преобразований. Это значит, что несмотря на отсутствие процедурной информации в программе, *неявно подразумевается наличие некоторой конкретной стратегии выполнения*. Таким образом, понятие «декларативности» относится в большей степени к восприятию программ человеком.

Кроме того, должны существовать определённые правила, касающиеся стратегии редукции, поскольку выбор различных β -редексов, как мы знаем, может оказать решающее влияние на завершимость. Как следствие, полностью определённый язык программирования из лямбда-исчисления получается лишь тогда, когда мы зададим эту стратегию. В дальнейшем мы увидим, какие решения были приняты в ходе проектирования различных функциональных языков, но перед этим нам придётся прервать наше изложение и обсудить введение понятия типа.

⁶ По всей видимости, Лэндин предпочитал термин «денотационный».

Глава 4

Типы

Типы представляют собой удобное средство определения различных разновидностей данных, наподобие логических и целочисленных значений либо функций. Благодаря типизации оказывается возможным гарантировать соблюдение ограничений, порождаемых этими различиями (например, что функция не должна применяться к аргументам с неподходящими типами). Что побуждает нас ввести понятие типа в лямбда-исчисление и языки программирования на его основе? Основания для такого решения можно найти как в логике, так и в программировании.

С точки зрения логики, требуется преодолеть парадокс Рассела, который препятствует попыткам построить непротиворечивое расширение лямбда-исчисления теорией множеств. Источником противоречий служит необычная циклическая природа используемого при этом приёма — применение функции к самой себе. Более того, если бы даже и не требовалось избежать парадокса, всё равно возникает интуитивное ощущение неясности формальной системы, в рамках которой разрешены подобные действия. Безусловно, самоприменение таких функций, как тождественная функция $(\lambda x. x)$ и функция-константа $(\lambda x. y)$, выглядит безобидно. В то же время, очевидна и потребность в более ясном описании того, какие именно семейства функций представимы в терминах лямбда-исчисления при условии, что нам точно известны области определения и значений этих функций, а также то, что мы их применяем лишь к аргументам, принадлежащим соответствующим областям определения. Введение Расселом типов в своей работе *Principia Mathematica* было продиктовано приведёнными соображениями.

Ещё одной причиной, которая побуждает нас подробно рассмотреть возможность расширения лямбда-исчисления понятием типа, является применение типизации в других языках программирования. Понятие типов данных встречается уже в языке FORTRAN, в котором различаются целые числа и числа с плавающей точкой. Причины появления типов в данном контексте не были связаны с изложенными ранее аргументами из области логики. Одной из таких причин была, очевидно, эффективность порождаемого компилятором кода. Наличие информации о допустимых способах использования той или иной переменной позволяет как генерировать более эффективный код, так и рациональнее распределять память. Например, реализация адресной арифметики в духе языка C должна учитывать размер объектов, к которым происходит обращение. Если p представляет собой указатель на объект размером 4 байта, то выражение $p + 1$ при трансляции в

машинный код на архитектурах с побайтовой адресацией памяти превращается в $p + 4$. Предшественник C, язык BCPL, был бестиповым, и в нём не делалось различий между целыми числами и указателями. Как следствие, соответствующие масштабирующие множители в каждой операции адресной арифметики задавались в программе явным образом, создавая тем самым существенные неудобства.

Дальнейшее развитие привело к тому, что типизация, оставаясь важным средством повышения эффективности, стала приобретать всё большее значение как инструмент ограниченной статической проверки корректности программ. Существенная доля ошибок, от очевидных опечаток до серьёзных концептуальных просчётов, проявляет себя нарушением правил типизации, благодаря чему эти ошибки могут быть выявлены непосредственно в ходе компиляции без запуска программы на исполнение. Более того, в ходе чтения исходных текстов типы зачастую играют роль документации. Наконец, типы данных могут применяться для улучшения модульности программ и скрытия информации при помощи таких определений различных структур данных, которые явно разделены на интерфейс и подробности реализации.

В то же время некоторые программисты выступают против использования типов, полагая, что для их стиля программирования ограничения, накладываемые типизацией, являются излишне утомительными. Как следствие, различается и уровень её поддержки языками программирования. Существуют бестиповые языки, как императивные (BCPL), так и функциональные (ISWIM, SASL и Erlang ???). Другие, подобно PL/I, имеют лишь *слабую типизацию*, которая допускает некоторые варианты совместного использования данных различных типов при помощи автоматических преобразований, реализуемых компилятором. Наконец, некоторые языки, такие как Lisp, осуществляют *динамический* контроль типов во время исполнения программы. Этот подход может, в принципе, существенно ухудшить производительность, поскольку требует дополнительных вычислительных ресурсов, подобно тому, как это происходит с другим известным источником накладных расходов — проверкой корректности обращений к массивам. Статическая же типизация, напротив, может существенно снизить издержки.¹

На практике важность тех или иных ограничений типизации существенно зависит от характера решаемых задач и стиля программирования. Разработка системы типов, обеспечивающей как возможность содержательного статического контроля, так и достаточный уровень гибкости, остаётся предметом активных исследований. Типизация, реализованная в языке ML, представляет собой важное достижение, поскольку в ней допускается *полиморфизм*, благодаря которому одна и та же функция может применяться к аргументам различных типов. Такой подход сохраняет все выгоды сильной статической типизации, дополняя их некоторыми возможностями, присущими слабому или динамическому контролю типов². Более того, программист как правило не обязан указывать типы явно — транслятор ML способен самостоятельно вывести наиболее общий тип каждого выражения, отвергая те из них, которые не поддаются типизации. Роль полиморфизма в процессе вывода типов будет рассмотрена далее. Таким образом, несомненно, что система типов языка ML делает его подходящим инструментом для широкого класса задач. Тем

¹Одним из направлений развития динамической типизации является её дополнение элементами статической, что зачастую позволяет добиться сравнимой эффективности. — Прим. перев.

²Возможно, также ценой дополнительных затрат.

не менее, мы не хотели бы создать у читателя ложного впечатления, что этот язык служит универсальным средством от всех проблем программирования.

4.1 Типизированное лямбда-исчисление

В первом приближении расширение лямбда-исчисления понятием типа не представляет особого труда, но в итоге, как будет показано, потребуется куда больше усилий. Основная идея состоит в том, что каждому терму назначается *тип*, после чего выражение $s\ t$, т. е. применение терма s к терму t , допустимо исключительно для совместимых типов, то есть в случае, когда типы s и t имеют вид $\sigma \rightarrow \tau$ и σ соответственно. Результирующий терм будет иметь при этом тип τ . Такую типизацию принято называть *сильной*.³ Терм t *обязан* иметь тип σ , подтипы и преобразования не допускаются. Такой подход составляет резкий контраст с некоторыми языками программирования, например, с языком C, в котором функция, ожидающая аргумент типа `float` либо `double`, принимает также значения типа `int`, выполняя автоматическое преобразование. Аналогичные понятия подтипов и преобразований возможно задать и в рамках лямбда-исчисления, но их освещение завело бы нас слишком далеко.

Введём для отношения « t имеет тип σ » обозначение $t : \sigma$. Подобная запись традиционно используется математиками при работе с функциональными пространствами, поскольку $f : \sigma \rightarrow \tau$ обозначает функцию f , отображающую множество σ во множество τ . Будем считать типы множествами, которые содержат соответствующие объекты, и трактовать $t : \sigma$ как $t \in \sigma$. Однако, несмотря на то, что мы предлагаем читателям также воспользоваться этой удобной аналогией, типизированное лямбда-исчисление будет в дальнейшем рассматриваться исключительно как формальная система, свободная от каких-либо интерпретаций.

4.1.1 Множество допустимых типов

Начнём формализацию строгим определением понятия типа. Предположим, что у нас имеется некоторое множество *примитивных типов*, в которое входят, например, типы `bool` и `int`. Составные типы могут быть определены при помощи *конструктора типа функции*. Формально, индуктивное определение множества типов Ty_C , основанного на множестве примитивных типов C , выглядит так:

$$\frac{\sigma \in C}{\sigma \in Ty_C}$$

$$\frac{\sigma \in Ty_C \quad \tau \in Ty_C}{\sigma \rightarrow \tau \in Ty_C}$$

Например, в рамках данного определения допустимы типы int , $bool \rightarrow bool$ либо $(int \rightarrow bool) \rightarrow int \rightarrow bool$. Будем считать операцию « \rightarrow » правоассоциативной, т. е. полагать выражение $\sigma \rightarrow \tau \rightarrow \upsilon$ равным $\sigma \rightarrow (\tau \rightarrow \upsilon)$. Такая трактовка естественно согласуется с другими синтаксическими правилами, касающимися каррирования.

³Сильную типизацию также часто называют *строгой*. — Прим. перев.

Следующим нашим шагом будет расширение системы типов в двух направлениях. Во-первых, введём наравне с примитивными типами (которые выполняют роль констант) так называемые *переменные типа*, которые впоследствии лягут в основу полиморфизма. Во-вторых, разрешим использование множества конструкторов других типов, помимо типа функции. Например, в дальнейшем нам понадобится конструктор \times для типа декартова произведения. Как следствие, наше индуктивное определение должно быть дополнено ещё одним выражением:

$$\frac{\sigma \in Ty_C \quad \tau \in Ty_C}{\sigma \times \tau \in Ty_C}$$

Поскольку язык ML допускает определение пользователем новых типов и их конструкторов, нам потребуется нотация, пригодная для описания произвольного множества конструкторов с произвольным количеством аргументов. Обозначим через $(\alpha_1, \dots, \alpha_n)con$ применение n -арного конструктора типа con к набору аргументов α_i . (Инфиксная форма будет использоваться лишь в некоторых широко известных частных случаях наподобие \rightarrow и \times .) Например, выражение $(\sigma)list$ трактуется как тип-список, все элементы которого имеют тип σ .

Принимая во внимание *free inductive generation*, можно доказать важное свойство типов, а именно, что $\sigma \rightarrow \tau \neq \sigma$. (В действительности справедливо более общее утверждение: тип не может равняться произвольному собственному подвыражению.) Это свойство исключает возможность применения терма к самому себе, за исключением случая, когда оба экземпляра терма, о которых идёт речь, имеют различные типы.

4.1.2 Типизация по Чёрчу и Карри

Известны два основных подхода к определению типизированного лямбда-исчисления. Один из них, разработанный Чёрчем, подразумевает *явное* указание типов. Каждому терму при этом назначается единственный тип. Другими словами, в ходе построения термов, каждому нетипизированному терму, которые были рассмотрены ранее, в дополнение указывается тип. Типы констант являются предопределёнными, но типы переменных могут быть произвольными. Точные правила построения типизированных термов приведены ниже:

$$\begin{array}{c} \hline v : \sigma \\ \hline \end{array}$$

$$\frac{\text{Константа } c \text{ имеет тип } \sigma}{c : \sigma}$$

$$\frac{s : \sigma \rightarrow \tau \quad t : \sigma}{s \ t : \tau}$$

$$\frac{v : \sigma \quad t : \tau}{\lambda v. t : \sigma \rightarrow \tau}$$

Однако, для наших целей лучше подходит *неявная* типизация, предложенная Карри. Структура термов соответствует нетипизированному случаю, при этом терм

может как иметь тип (причём не один), так и не иметь его.⁴ Например, тождественной функции $\lambda x. x$ может быть вполне обоснованно назначен произвольный тип вида $\sigma \rightarrow \sigma$. Применительно к языку ML существуют два взаимосвязанных довода в пользу данного подхода к типизации. Во-первых, он позволяет удобнее выразить присущий ML полиморфизм, а во-вторых, хорошо согласуется с практикой программирования на этом языке, в ходе которого не требуется задавать типы явно.

В то же время, некоторые формальные аспекты назначения типов по Карри оказываются достаточно сложными. Отношение типизируемости не может быть задано в отрыве от некоторого *контекста*, представляющего собой конечное множество утверждений относительно типов переменных. Обозначим через

$$\Gamma \vdash t : \sigma$$

утверждение «в контексте Γ терму t может быть назначен тип σ ». (Если это утверждение справедливо при пустом контексте, выражение сокращается до $\vdash t : \sigma$ или даже до $t : \sigma$.) Элементы множества Γ имеют вид $v : \sigma$ т. е. сами по себе являются утверждениями относительно типов отдельных переменных, обычно тех, которые входят в терм t . Будем полагать, что контекст Γ не содержит противоречивых утверждений о типе некоторой переменной; при желании, мы можем рассуждать о нём как о частичной функции, отображающей индексное множество переменных во множество типов. Использование нами символа \vdash соответствует его роли в традиционной логике, где $\Gamma \vdash \phi$ принято трактовать как «утверждение ϕ следует из множества посылок Γ ».

4.1.3 Формальные правила типизации

Формулировка правил назначения типов выражениям достаточно естественна. Прежде, чем мы приведём эти правила, напомним ещё раз, что $t : \sigma$ следует интерпретировать как « t может иметь тип σ ».

$$\frac{v : \sigma \in \Gamma}{\Gamma \vdash v : \sigma}$$

$$\frac{\text{Константа } c \text{ имеет тип } \sigma}{c : \sigma}$$

$$\frac{\Gamma \vdash s : \sigma \rightarrow \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash s t : \tau}$$

$$\frac{\Gamma \cup \{v : \sigma\} \vdash t : \tau}{\Gamma \vdash \lambda v. t : \sigma \rightarrow \tau}$$

Ещё раз повторим, что эти выражения следует понимать как индуктивное определение отношения типизируемости, так что терм может иметь тип лишь тогда, когда последний выводим при помощи упомянутых выше правил. В качестве примера

⁴Поборники чистоты терминологии могут возразить против использования в данном случае термина «типизированное лямбда-исчисление», считая более подходящим «нетипизированное лямбда-исчисление, дополненное понятием назначения типа»

рассмотрим процедуру вывода типа тождественной функции. Согласно правилу типизации переменных, мы имеем:

$$\{x : \sigma\} \vdash x : \sigma$$

откуда, применив последнее правило, получаем:

$$\emptyset \vdash \lambda x. x : \sigma \rightarrow \sigma$$

Применив установленное ранее соглашение о пустых контекстах, мы можем сократить это выражение до $\lambda x. x : \sigma \rightarrow \sigma$. На данном примере также хорошо видна как важная роль контекстов в типизации по Карри, так и их необязательность в рамках типизации по Чёрчу. Опуская контекст, мы можем вывести $x : \tau$ для произвольного типа τ , после чего, согласно последнему правилу, получаем $\lambda x. x : \sigma \rightarrow \tau$ — налицо различие с интуитивной трактовкой тождественной функции! Эта проблема не возникает в ходе типизации по Чёрчу, поскольку в её рамках либо обе переменные имеют тип σ , откуда получаем $\lambda x. x : \sigma \rightarrow \sigma$, либо эти переменные на самом деле различны (поскольку различны их типы, которые считаются неотъемлемой частью терма). В последнем случае тип выражения в действительности будет равен $\lambda x : \sigma. (x : \tau) : \sigma \rightarrow \tau$, но это обосновано тем, что данное выражение альфа-эквивалентно $\lambda x : \sigma. (y : \tau) : \sigma \rightarrow \tau$. Поскольку в ходе типизации по Карри термы не содержат в себе типов явно, нам требуется некоторый механизм связывания между собой одинаковых переменных.

4.1.4 Сохранение типа

Очевидное сходство структуры термов типизированного и нетипизированного лямбда-исчисления порождает естественное желание применить в типизированном случае аппарат формальных преобразований, разработанный для нетипизированного. Однако, нам потребуется предварительно доказать, что тип выражения в ходе преобразований не изменяется (такое свойство называется *сохранением типа*). Убедиться в этом не представляет труда; мы рассмотрим краткое изложение доказательства для случая η -преобразования, предоставив остальные читателю в качестве упражнения. Прежде всего, докажем две леммы, которые весьма очевидны, но тем не менее, требуют формального обоснования. Во-первых, покажем, что добавление новых элементов в контекст не влияет на типизируемость:

Лемма 4.1 (О монотонности) *Если $\Gamma \vdash t : \sigma$ и $\Gamma \subseteq \Delta$, то справедливо $\Delta \vdash t : \sigma$.*

Доказательство: Применим индукцию по структуре t . Зафиксировав t , докажем приведённое выше утверждение для всевозможных Γ и Δ , поскольку в ходе шага индукции для абстракций эти множества изменяются. Если t — переменная, справедливо $t : \sigma \in \Gamma$, из чего следует и $t : \sigma \in \Delta$, откуда получаем требуемое. Если t — константа, желаемый вывод очевиден, поскольку множество констант и их типов не зависит от контекста. В случае терма t , имеющего вид комбинации термов s и u , для некоторого типа τ выполняется $\Gamma \vdash s : \tau \rightarrow \sigma$ и $\Gamma \vdash u : \tau$. Согласно индуктивному предположению, $\Delta \vdash s : \tau \rightarrow \sigma$ и $\Delta \vdash u : \tau$, откуда также получаем требуемое. Наконец, если терм t представляет собой абстракцию $\lambda x. s$, то согласно последнему правилу типизации σ имеет вид $\tau \rightarrow \tau'$, а также справедливо, что $\Gamma \cup$

$\{x : \tau\} \vdash s : \tau'$. Так как $\Gamma \subseteq \Delta$, мы получаем $\Gamma \cup \{x : \tau\} \subseteq \Delta \cup \{x : \tau\}$, откуда по индуктивному предположению $\Delta \cup \{x : \tau\} \vdash s : \tau'$. Применяя правило типизации абстракций, получаем требуемое. \square

Во-вторых, элементы контекста, представляющие переменные, которые не являются свободными в заданном терме, могут игнорироваться.

Лемма 4.2 Если $\Gamma \vdash t : \sigma$, то справедливо также $\Gamma_t \vdash t : \sigma$, где Γ_t содержит исключительно свободные переменные терма t ($\Gamma_t = \{x : \alpha \mid x : \alpha \in \Gamma \text{ и } x \in FV(t)\}$).

Доказательство: Аналогично предыдущей лемме, докажем наше утверждение для произвольного контекста Γ и соответствующего ему Γ_t путём структурной индукции по t . Если t — переменная, то $\Gamma \vdash t : \sigma$ требует наличия в контексте элемента $x : \sigma$. Согласно первому правилу типизации $\{x : \sigma\} \vdash x : \sigma$, что и требуется. Тип константы не зависит от контекста, так что лемма справедлива и в этом случае. Если терм t представляет собой комбинацию термов вида s и u , то для некоторого τ справедливо $\Gamma \vdash s : \tau \rightarrow \sigma$ и $\Gamma \vdash u : \tau$. Согласно индуктивному предположению, $\Gamma_s \vdash s : \tau \rightarrow \sigma$ и $\Gamma_u \vdash u : \tau$. Согласно лемме о монотонности, получаем $\Gamma_{su} \vdash s : \tau \rightarrow \sigma$ и $\Gamma_{su} \vdash u : \tau$, поскольку $FV(s \ u) = FV(s) \cup FV(u)$. Применив правило вывода типа комбинации термов, получаем $\Gamma_{su} \vdash t : \sigma$. Наконец, если t имеет вид $\lambda x. s$, это подразумевает $\Gamma \cup \{x : \tau\} \vdash s : \tau'$, где σ имеет форму $\tau \rightarrow \tau'$. Согласно индуктивному предположению, $(\Gamma \cup \{x : \tau\})_s \vdash s : \tau'$, откуда $(\Gamma \cup \{x : \tau\})_s - \{x : \tau\} \vdash (\lambda x. s) : \sigma$. Теперь нам требуется лишь отметить, что $(\Gamma \cup \{x : \tau\})_s - \{x : \tau\} \subseteq \Gamma_t$ и ещё раз применить лемму о монотонности. \square

Приступим к доказательству основного результата этого раздела.

Теорема 4.3 (О сохранении типа) Если $\Gamma \vdash t : \sigma$ и $t \xrightarrow{\eta} t'$, то из этого следует, что $\Gamma \vdash t' : \sigma$.

Доказательство: Поскольку по условию теоремы терм t является η -редексом, он должен иметь структуру $(\lambda x. t \ x)$, причём $x \notin FV(t)$. Следовательно, его тип может быть выведен лишь из последнего правила типизации, при этом σ имеет вид $\tau \rightarrow \tau'$, и справедливо $\{x : \tau\} \vdash (t \ x) : \tau'$. Дальнейший анализ требует применения правила вывода типа комбинаций. Поскольку контекст может содержать не более одного утверждения о типе каждой переменной, справедливо $\{x : \tau\} \vdash t : \tau \rightarrow \tau'$. Так как по условию $x \notin FV(t)$, то применив лемму 4.2, получаем $\vdash t : \tau \rightarrow \tau'$, что и требовалось. \square

Собрав воедино результаты аналогичных доказательств для других преобразований, получаем, что если $\Gamma \vdash t : \sigma$ и $t \xrightarrow{\eta} t'$, то выполняется также $\Gamma \vdash t' : \sigma$. Важность этого вывода в том, что если бы правила вычислений, применяемые в ходе исполнения программы, могли изменять типы выражений, это подорвало бы основы статической типизации.

4.2 Полиморфизм

Типизация по Карри предоставляет в наше распоряжение разновидность полиморфизма, позволяя назначить заданному терму различные типы. Следует различать схожие понятия полиморфизма и перегрузки. Оба они подразумевают,

что выражение может иметь множество типов. Однако, в случае полиморфизма все эти типы структурно связаны друг с другом, так что допустимы любые из них, удовлетворяющие заданному образцу. Например, тождественной функции можно назначить тип $\sigma \rightarrow \sigma$, или $\tau \rightarrow \tau$, либо даже $(\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau)$, но все они имеют одинаковую структуру. С другой стороны, суть перегрузки в том, что заданная функция может иметь различные типы, структура которых может различаться, либо же допустимо лишь ограниченное множество типов. Например функции $+$ может быть позволено иметь тип $int \rightarrow int \rightarrow int$ либо $float \rightarrow float \rightarrow float$, но не $bool \rightarrow bool \rightarrow bool$.⁵ Ещё одним близким понятием являются подтипы, представляющие собой более жёсткую форму перегрузки. Введение подтипов позволяет трактовать некоторый тип как подмножество другого. Однако, этот подход на практике оказывается куда сложнее, чем кажется на первый взгляд.⁶

4.2.1 Проблемы let-полиморфизма

К сожалению, определённая выше система типов накладывает некоторые нежелательные ограничения на полиморфизм. Например, следующее выражение абсолютно корректно:

$$\text{if } (\lambda x. x) \text{ true then } (\lambda x. x) \text{ 1 else 0}$$

Докажем, что это выражение может быть типизировано согласно нашим правилам. Предположим, что константам можно назначить типы в пустом контексте, и что мы можем двукратно применить правило типизации комбинации термов для назначения типа **if** (принимая во внимание, что выражение вида $\text{if } b \text{ then } t_1 \text{ else } t_2$ является всего лишь сокращённой записью для $\text{COND } b \ t_1 \ t_2$).

$$\frac{\frac{\frac{\{x : bool\} \vdash x : bool}{\vdash (\lambda x. x) : bool \rightarrow bool} \quad \vdash \text{true} : bool}{\vdash (\lambda x. x) \text{ true} : bool} \quad \frac{\frac{\{x : int\} \vdash x : int}{\vdash (\lambda x. x) : int \rightarrow int} \quad \vdash 1 : int}{\vdash (\lambda x. x) \text{ 1} : int} \quad \vdash 0 : int}{\vdash \text{if } (\lambda x. x) \text{ true then } (\lambda x. x) \text{ 1 else 0} : int}$$

Два экземпляра тождественной функции получают типы $bool \rightarrow bool$ и $int \rightarrow int$ соответственно. Далее рассмотрим другое выражение:

$$\text{let } I = \lambda x. x \text{ in if } I \text{ true then } I \text{ 1 else 0}$$

Согласно нашим определениям, это всего лишь удобный способ записи для

$$(\lambda I. \text{if } I \text{ true then } I \text{ 1 else 0}) (\lambda x. x)$$

Нетрудно убедиться, что тип этого выражения не может быть выведен в рамках наших правил. Мы имеем *единственный* экземпляр тождественной функции,

⁵Стрейчи, которому принадлежит авторство понятия полиморфизма, использовал термины *параметрический* и *ad hoc полиморфизм* вместо принятых в данном пособии терминов *полиморфизм* и *перегрузка* соответственно.

⁶Например, если тип α является подтипом α' , должен ли тип $\alpha \rightarrow \beta$ считаться подтипом $\alpha' \rightarrow \beta$ или наоборот? В зависимости от конкретной ситуации, более предпочтительной оказывается либо первая, либо вторая интерпретация («ковариантность» либо «контравариантность» типов).

которому должны назначить единственный тип. Подобное ограничение на практике неприемлемо, поскольку функциональное программирование предполагает частое использование `let`. Если правила типизации не будут изменены, многие выгоды полиморфизма окажутся потерянными. Нашим решением будет отказ от трактовки конструкции `let` как сокращённой записи в пользу реализации её как примитива языка, после чего ко множеству правил типизации следует добавить новое правило:

$$\frac{\Gamma \vdash s : \sigma \quad \Gamma \vdash t[s/x] : \tau}{\Gamma \vdash \text{let } x = s \text{ in } t : \tau}$$

Это правило, которым вводится понятие *let-полиморфизма*, демонстрирует, что по крайней мере с точки зрения типизации, `let`-связанные переменные трактуются как простые подстановки соответствующих выражений вместо их имён. Дополнительная посылка $\Gamma \vdash s : \sigma$ требуется исключительно для того, чтобы гарантировать существование корректного типа выражения s , причём точное значение этого типа нас не интересует. Цель данного ограничения в том, чтобы избежать ошибочных выводов о существовании корректных типов для таких термов, как

$$\text{let } x = \lambda f. f \text{ in } 0$$

Теперь мы в состоянии вывести тип нашего проблемного выражения, пользуясь приведёнными выше правилами:

$$\frac{\frac{\overline{\{x : \sigma\} \vdash x : \sigma} \quad \text{См. выше}}{\vdash \lambda x. x : \sigma \rightarrow \sigma} \quad \vdash \text{if } (\lambda x. x) \text{ true then } (\lambda x. x) \text{ 1 else } 0 : \text{int}}{\vdash \text{let } I = \lambda x. x \text{ in if } I \text{ true then } I \text{ 1 else } 0 : \text{int}}$$

4.2.2 Наиболее общий тип

Как было сказано ранее, тип некоторых выражений, таких как $\lambda f. f \text{ } f$ либо $\lambda f. (f \text{ true}, f \text{ 1})$, вывести невозможно. Типизируемые выражения обычно имеют множество типов, хотя некоторые из них, например, *true* — в точности один. Мы уже упоминали, что разновидность полиморфизма, доступная в языке ML, называется *параметрической*, т. е. всевозможные типы выражения должны обладать структурным подобием. Более того, для каждого типизируемого выражения существует так называемый *наиболее общий (главный) тип*, причём все возможные типы данного выражения представляют собой экземпляры наиболее общего типа. Прежде, чем изложить этот результат формально, введём некоторые термины.

Начнём с расширения определения типа, дополнив его понятием *переменной типа*. Это значит, что типы могут быть построены путём применения конструкторов типов как к типам-константам, так и к переменным. Будем использовать греческие буквы α и β для обозначения переменных типа, а σ и τ — произвольных типов. При помощи этой расширенной нотации мы в состоянии определить понятие подстановки типа в другой тип вместо переменной типа. Такая подстановка совершенно аналогична подстановке термов, так что мы даже будем использовать те же самые обозначения (например, $(\sigma \rightarrow \text{bool})[(\sigma \rightarrow \tau)/\sigma] = (\sigma \rightarrow \tau) \rightarrow \text{bool}$). Однако, формальное определение подстановки типов проще, чем для термов, так как не требует учёта связывания переменных. Для удобства дальнейшего изложения

расширим его на случай множественной параллельной подстановки:

$$\begin{aligned}\alpha_i[\tau_1/\alpha_1, \dots, \tau_n/\alpha_k] &= \tau_i \text{ if } \alpha_i \neq \beta \text{ for } 1 \leq i \leq k \\ \beta[\tau_1/\alpha_1, \dots, \tau_n/\alpha_k] &= \beta \text{ if } \alpha_i \neq \beta \text{ for } 1 \leq i \leq k \\ (\sigma_1, \dots, \sigma_n)con[\theta] &= (\sigma_1[\theta], \dots, \sigma_n[\theta])con\end{aligned}$$

Чтобы не загромождать определение, мы трактуем типы-константы как нуль-арные конструкторы типов, т. е. считаем $()int$ эквивалентным int ; при желании можно легко вернуться к прежним обозначениям явным добавлением соответствующих частных случаев. Опираясь на приведённое определение подстановки, введём отношение «тип σ является *более общим*, чем тип σ' », обозначив его через $\sigma \preceq \sigma'$.⁷ Пара типов входит в данное отношение тогда и только тогда, когда найдётся множество подстановок θ такое, что $\sigma' = \sigma\theta$. Например:

$$\begin{aligned}\alpha &\preceq \sigma \\ \alpha \rightarrow \alpha &\preceq \beta \rightarrow \beta \\ \alpha \rightarrow bool &\preceq (\beta \rightarrow \beta) \rightarrow bool \\ \beta \rightarrow \alpha &\preceq \alpha \rightarrow \beta \\ \alpha \rightarrow \alpha &\not\preceq (\beta \rightarrow \beta) \rightarrow \beta\end{aligned}$$

С учётом изложенного выше, сформулируем основную теорему данного раздела:

Теорема 4.4 *Каждый типизируемый терм имеет главный тип, т. е. для произвольного $t : \tau$ найдется тип σ такой, что $t : \sigma$ и для любого типа σ' из $t : \sigma'$ следует $\sigma \preceq \sigma'$.*

Легко убедиться, что отношение \preceq является отношением квазипорядка, то есть, рефлексивно и транзитивно. Главный тип не уникален, но при этом все его возможные формы равны с точностью до переименования входящих в них переменных типа. В более точной формулировке, если σ и τ являются одновременно главными типами выражения, то справедливо $\sigma \sim \tau$, то есть $\sigma \preceq \tau$ и $\tau \preceq \sigma$ одновременно.

Доказательство теоремы о главном типе не требует особых усилий, но ввиду большого объёма в рамках данного курса не рассматривается. Следует лишь запомнить его основное свойство: процедура доказательства представляет собой конкретный алгоритм поиска такого типа. Этот алгоритм известен как *алгоритм Милнера*, либо, чаще, как алгоритм Хиндли-Милнера.⁸ Все реализации ML и некоторых других функциональных языков включают в себя некоторую разновидность этого алгоритма, благодаря чему для произвольных выражений автоматически выводится их главный тип либо устанавливается невозможность корректной типизации.

⁷Это отношение рефлексивно, так что формулировка «не менее общий, чем» была бы точнее.

⁸Приведённая нами формулировка теоремы была опубликована в работе [Milner \(1978\)](#), но как сама теорема, так и алгоритм были ранее открыты Хиндли в его исследованиях по комбинаторной логике.

4.3 Сильная нормализация

Обратимся вновь к нашим примерам термов, не имеющих нормальной формы, таким как

$$\begin{aligned} & ((\lambda x. x x x) (\lambda x. x x x)) \\ \longrightarrow & ((\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x)) \\ \longrightarrow & (\dots) \end{aligned}$$

В рамках типизированного лямбда-исчисления подобная ситуация невозможна на основании теоремы о *сильной нормализации*, доказательство которой слишком длинно, чтобы быть здесь приведённым.

Теорема 4.5 (О сильной нормализации) *Любой типизируемый терм имеет нормальную форму, а любая возможная последовательность редукций, которая начинается с типизируемого терма, завершается.*⁹

На первый взгляд преимущества очевидны — функциональная программа, удовлетворяющая нашей дисциплине типов, может вычисляться в произвольном порядке, при этом процесс редукции всегда конечен и приводит к единственной нормальной форме. (Единственность следует из теоремы Чёрча-Россера, которая остаётся справедливой и в случае типизированного лямбда-исчисления.) Однако, возможность реализации незавершённых функций необходима для обеспечения Тьюринг-полноты,¹⁰ в противном случае мы более не в состоянии определить произвольные вычислимые функции, более того — даже не всё множество всюду определённых функций.

Мы бы могли пренебречь этим ограничением, если бы оно позволяло нам использовать все функции, представляющие практический интерес. Однако, это не так — класс всевозможных функций, представимых в рамках типизированного лямбда-исчисления, оказывается весьма узким. В работе [Schwichtenberg \(1976\)](#) показано, что класс представимых функций на основе нумералов Чёрча ограничен всевозможными полиномами либо кусочными функциями на их основе. Отметим, что это *strictly intensional*: рассмотрев другое представление чисел, получим другой класс функций. В любом случае, для универсального языка программирования этого недостаточно.

Поскольку все определимые функции являются всюду определёнными, мы, очевидно, не в состоянии давать произвольные рекурсивные определения. (???) В самом деле, оказывается, что обычные комбинаторы неподвижной точки не поддаются типизации; очевидно, что тип $Y = \lambda f. (\lambda x. f(x x))(\lambda x. f(x x))$ не существует, поскольку x применяется к самому себе, будучи связанным лямбда-абстракцией. Для восстановления Тьюринг-полноты введём альтернативный способ задания произвольных рекурсивных функций, не принося в жертву типизацию.

⁹Под «слабой нормализацией» подразумевается первая часть данного утверждения о возможности преобразования произвольного терма в нормальную форму; при этом некоторые последовательности редукций могут оказаться бесконечными.

¹⁰Располагая любым рекурсивным перечислением множества всюду определённых вычислимых функций, мы всегда можем определить ещё одну, не входящую в это множество, методом диагонализации. (Подробнее об этом будет рассказано в курсе теории алгоритмов.)

Определим полиморфный оператор рекурсии, всевозможные типы которого имеют вид

$$Rec : ((\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau)) \rightarrow \sigma \rightarrow \tau$$

и дополнительное правило редукции, согласно которому для произвольной функции $F : (\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau)$ мы имеем

$$Rec F \longrightarrow F (Rec F)$$

Начиная с этого момента будем полагать, что рекурсивные определения вида `let rec` отображаются на эти операторы рекурсии.

Дополнительная литература

Типизированное лямбда-исчисление рассматривается, в числе прочего, в работах [Barendregt \(1984\)](#) и [Hindley and Seldin \(1986\)](#). Основополагающая работа [Milner \(1978\)](#) продолжает оставаться важным источником информации о полиморфной типизации, включая алгоритм поиска главного типа. Хорошее введение в типизированное лямбда-исчисление, включающее доказательство теоремы о сильной нормализации, а также обсуждение некоторых интересных взаимосвязей с логикой, даётся в [Girard, Lafont, and Taylor \(1989\)](#). В данной работе также обсуждается более развитая версия типизированного лямбда-исчисления под названием System F, в рамках которой возможно определение большинства требуемых на практике функций даже при сохранении свойства сильной нормализации.

Упражнения

1. Справедливо ли, что из $\Gamma \vdash t : \sigma$ для произвольной подстановки θ следует $\Gamma \vdash t : (\sigma\theta)$?
2. Докажите формально теорему 4.3 о сохранении типа для α и β -преобразования.
3. Покажите, что свойство сохранения типа необратимо, т. е. что возможна ситуация, когда из справедливости $t \longrightarrow t'$ и $\Gamma \vdash t' : \sigma$ не следует, что $\Gamma \vdash t : \sigma$.
4. (*) Докажите, что каждый терм типизированного лямбда-исчисления, чей *главный* тип равен $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$, редуцируется до нумерала Чёрча.
5. (*) В какой мере процесс проверки типов является обратимым, т. е. допускает вывод терма по его типу? Например, справедливо ли, что в чистом типизированном лямбда-исчислении с переменными типа, но без констант и оператора рекурсии, любое $t : \alpha \rightarrow \beta \rightarrow \alpha$ на самом деле эквивалентно $K = \lambda x y. x$ в традиционной трактовке лямбда-эквивалентности. Если это так, то каковы дальнейшие возможности обобщения результата?¹¹
6. (*) Будем говорить, что некоторое отношение редукции \longrightarrow *обладает слабым свойством Чёрча-Россера*, если всякий раз, когда $t \longrightarrow t_1$

¹¹Подробнее по данному вопросу см. [Mairson \(1991\)](#).

и $t \longrightarrow t_2$, найдётся u такое, что $t_1 \longrightarrow^* u$ и $t_2 \longrightarrow^* u$, где \longrightarrow^* представляет собой рефлексивное транзитивное замыкание \longrightarrow . Докажите лемму Ньюмена, в которой утверждается, что если отношение обладает слабым свойством Чёрча-Россера и удовлетворяет принципам сильной нормализации, то для \longrightarrow^* справедливо свойство Чёрча-Россера. (Указание: воспользуйтесь трансфинитной индукцией.)¹²

¹²Если в ходе доказательства возникнут затруднения, см. работу [Huet \(1980\)](#).

Глава 5

Знакомство с ML

В предыдущих главах мы начали с чистого λ -исчисления, которое затем систематически расширяли новыми возможностями. Например, мы добавили примитивную конструкцию `let`, чтобы сделать полиморфную типизацию более полезной, а также оператор рекурсии для восстановления вычислительной полноты, потерянной после введения типов. Двигаясь дальше по этому пути, мы в конечном счёте приходим к ML, при этом сохранив простоту мировоззрения, сформированного в ходе изучения типизированного λ -исчисления.

Очередным этапом на этом пути будет отказ от представления данных (например, натуральных чисел и логических значений) в виде термов лямбда-исчисления и замена их примитивами, такими как типы `bool` (для логических значений) и `int` (для целых чисел). В дополнение, введём новые конструкторы типов, такие как \times , — желательность подобного шага уже упоминалась в предыдущей главе. С этими изменениями также связаны новые константы и новые правила преобразования. Например, выражение $2 + 2$ будет вычисляться с использованием машинной арифметики вместо представления его в нумералах Чёрча и выполнения β -преобразований. Эти дополнительные преобразования, рассматриваемые как расширение обычных λ -операций, часто называются ‘ δ -преобразованиями’. В течение курса мы неоднократно увидим, чем язык ML отличается от чистого λ -исчисления. Первым делом обратимся к фундаментальному вопросу стратегии вычисления выражений в ML.

5.1 Энергичное вычисление

Как уже было сказано ранее, с теоретической точки зрения нормальный порядок (сверху вниз, слева направо) редукции выражений более предпочтителен, потому что если хоть какая-то стратегия завершается, то и эта тоже.¹ Однако, такой подход имеет некоторые практические недостатки. Например, рассмотрим следующее выражение:

$$(\lambda x. x + x + x) (10 + 5)$$

При использовании нормального порядка редукции мы получаем $(10 + 5) + (10 +$

¹Данная стратегия подобна некоторым, используемым в традиционных языках, таких как Algol 60, где её называют *вызов по имени*.

5) + (10 + 5), так что на следующих шагах мы должны вычислить одно и то же выражение трижды. На практике это совершенно недопустимо. Существуют два основных решения данной проблемы, и эти решения делят мир функционального программирования на два лагеря.

Первое решение — придерживаться нормального порядка редукции, но при этом пытаться оптимизировать реализацию так, чтобы разнообразные подвыражения, возникающие при таком подходе, использовались совместно и никогда не вычислялись более одного раза. В самой реализации выражения представляются в виде ориентированных ациклических графов, а не в виде деревьев. Этот подход известен как *ленивое* (*lazy*) или *вызов по необходимости* (*call-by-need*) вычисление, поскольку выражения вычисляются только тогда, когда необходимо.

Вторым решением является попытка перевернуть с ног на голову теоретические размышления о стратегии редукции и вычислять аргументы функции до начала её собственного вычисления. Этот подход известен как *аппликативный порядок* или *энергичное* вычисление. Последнее имя возникло из-за того, что аргументы функции вычисляются даже тогда, когда они не нужны, например, t в $(\lambda x. y) t$. Конечно, применение аппликативного порядка означает, что процесс вычисления некоторых выражений может заикливаться, тогда как он завершается при работе в ленивом режиме. Но это считается допустимым, поскольку подобных ситуаций достаточно легко избежать на практике. В любом случае, стратегия энергичного вычисления является стандартной для многих языков программирования, таких как C, где её называют *вызов по значению* (*call by value*).

ML использует энергичное вычисление по двум основным причинам. Управление редукцией и совместным использованием подвыражений, которое требуется при ленивом вычислении, является достаточно сложным, и реализация может быть относительно неэффективной и трудной. Если программист не проявит должной осторожности, то память может заполниться невычисленными выражениями, так что в общем случае оценить потребление памяти программой будет затруднительно. В действительности, многие реализации ленивого вычисления стараются оптимизировать его путём использования энергичного вычисления в тех местах, где семантика не отличается.² В противоположность этому, в ML мы всегда сначала вычисляем аргументы функций, и только затем выполняем β -редукцию — это просто, эффективно и легко реализуется с использованием стандартных технологий построения компиляторов.

Второй причиной для выбора аппликативного порядка вычислений служит то, что ML не является *чистым* функциональным языком, а имеет императивные возможности (переменные, присваивание и т.п.). Следовательно, порядок вычисления подвыражений может изменить *семантику*, а не просто влияет на эффективность. Если используется ленивое вычисление, то для программиста становится практически невозможным отчётливо представить (в нетривиальной программе) когда и какое подвыражение вычисляется. С другой стороны, в энергичной системе, подобной ML, достаточно лишь помнить соответствующие простые правила.

Однако, важно осознавать, что стратегия вычислений ML *не* просто редукция снизу вверх, в противоположность нормальному порядку. В действительности ML

²Они стараются выполнить *анализ строгости* — один из видов статического анализа, который часто помогает определить, что аргументы должны быть вычислены (Mycroft 1981).

никогда не вычисляет содержимое λ -абстракций. (В частности, он никогда не редуцирует η -редексы, а только β -редексы.) При вычислении $(\lambda x. s[x]) t$, сначала вычисляется t . Однако $s[x]$ не затрагивается, поскольку оно является содержимым λ -абстракции. Кроме того, любое подвыражение t , которое является содержимым λ -абстракции, также остаётся нетронутым. Вот точные правила вычисления:

- Константы вычисляются сами в себя.
- Вычисления заканчиваются на λ -абстракциях, и не затрагивают их содержимого. В частности, не выполняется η -преобразование.
- При вычислении комбинации $s t$ сначала вычисляются оба терма, s и t . Потом, при условии что результатом вычисления s является λ -абстракция, производится самое внешнее β -преобразование, и процесс повторяется.

Порядок вычисления s и t отличается в зависимости от версии ML. В той версии, которую мы будем использовать, сначала всегда вычисляется t . Строго говоря, мы также должны задать правило для **let**-выражений, поскольку, как упоминалось, они теперь считаются примитивами. Однако, с точки зрения стратегии вычислений, они как и прежде могут рассматриваться как применение λ -абстракции к аргументу, который будет вычислен первым. Для того, чтобы сделать это явным, правило для

$$\text{let } x = s \text{ in } t$$

гласит, что сначала вычисляются все s , а результат подставляется вместо x в t , после чего вычисляется новое значение t . Рассмотрим некоторые примеры вычисления выражений:

$$\begin{aligned} (\lambda x. (\lambda y. y + y) x)(2 + 2) &\longrightarrow (\lambda x. (\lambda y. y + y) x)4 \\ &\longrightarrow (\lambda y. y + y)4 \\ &\longrightarrow 4 + 4 \\ &\longrightarrow 8 \end{aligned}$$

Заметим, что подтерм $(\lambda y. y + y) x$ не редуцируется, поскольку он находится в пределах λ -абстракции. Однако, редуцируемые термы, не находящиеся внутри λ -абстракций обеих функций, а также аргумент, редуцируются до того, как вычисляется применение функции, например, второй шаг будет следующим:

$$\begin{aligned} ((\lambda f x. f x) (\lambda y. y + y)) (2 + 2) &\longrightarrow ((\lambda f x. f x) (\lambda y. y + y)) 4 \\ &\longrightarrow (\lambda x. (\lambda y. y + y) x) 4 \\ &\longrightarrow (\lambda y. y + y) 4 \\ &\longrightarrow 4 + 4 \end{aligned}$$

Тот факт, что ML не вычисляет содержимое λ -абстракций, является ключевым для опытных программистов на ML. Он даёт возможность точного контроля над вычислением выражений и может использоваться для имитации многих полезных возможностей ленивой стратегии. Простейшие примеры этого будут рассмотрены в следующем разделе.

5.2 Результаты энергичного вычисления

Использование энергичного вычисления заставляет нас объявить примитивами некоторые дополнительные языковые конструкции, определив для них специальные методы редукции, вместо того, чтобы реализовать их напрямую в терминах λ -исчисления. В частности, мы не можем больше рассматривать условную конструкцию

if b then e_1 else e_2

как применение обычного трехкомпонентного (тернарного) оператора

COND $b\ e_1\ e_2$

Причина заключается в том, что по правилам энергичного вычисления мы всегда должны вычислить все выражения, b , e_1 и e_2 , до вычисления COND. Как правило, последствия этого фатальны. Например, заново рассмотрим наше определение функции вычисления факториала:

let rec fact(n) = if ISZERO n then 1 else $n * \text{fact}(\text{PRE } n)$

Если условное выражение будет вычислять все свои аргументы, то при вычислении fact(0), ветвь ‘else’ также должна быть вычислена, что в свою очередь вызывает вычисление fact(PRE 0). Но это также потребует вычисления fact(PRE (PRE 0)), и т.д. Соответственно, вычисление превратится в бесконечный цикл.

Таким образом, мы делаем условное выражение примитивной конструкцией и меняем обычную стратегию редукции так, что *сначала* вычисляется логическое выражение, а затем — *только одна* соответствующая ветвь условия.

А что происходит с самим процессом рекурсии? Мы предложили интерпретацию рекурсивных определений в терминах рекурсивного оператора *Rec* с его собственным правилом редукции:

$\text{Rec } f \longrightarrow f(\text{Rec } f)$

который также будет заикливаться при использовании стратегии энергичного вычисления:

$\text{Rec } f \longrightarrow f(\text{Rec } f) \longrightarrow f(f(\text{Rec } f)) \longrightarrow f(f(f(\text{Rec } f))) \longrightarrow \dots$

Однако, достаточно лишь очень простого изменения правил редукции, чтобы решить эту проблему:

$\text{Rec } f \longrightarrow f(\lambda x. \text{Rec } f\ x)$

Теперь λ -абстракция в правой части правила означает, что $\lambda x. \text{Rec } f\ x$ вычисляется само в себя (то есть, не вычисляется вовсе), и только после того, как выражение было редуцировано в ходе подстановки этой λ -абстракции в терм f , вычисление продолжается.

5.3 Семейство языков ML

Мы говорили о ‘ML’ так, как будто это один язык. На самом деле существует много вариантов ML, в том числе и ‘Lazy ML’ — реализация университета Chalmers в Швеции, которая базируется на ленивых вычислениях. Наиболее популярная версия ML в образовании — это ‘Standard ML’, но мы будем использовать другую, которая называется CAML (‘camel’) Light.³ Мы выбрали CAML Light по следующим причинам:

- реализация имеет небольшой объём и хорошо переносима между платформами, так что она эффективно работает на Unix, PC, Mac, и других.
- Система очень проста синтаксически и семантически, что делает её достаточно простой для изучения.
- Система хорошо подходит для практического использования. Например, она имеет интерфейс к библиотекам на языке C и поддерживает стандартную отдельную компиляцию, совместимую с `make`.

Однако, мы будем изучать достаточно общие техники, так что любой написанный код может быть запущен (с небольшими синтаксическими изменениями) на любой версии ML, и часто, на других функциональных языках.

5.4 Запуск ML

ML уже установлен на рабочий сервер (Thor). Для того, чтобы использовать его, вам необходимо добавить каталог с исполняемыми файлами CAML в переменную среды `PATH`. Это может быть сделано следующим образом (предполагая, что вы используете командный процессор `bash` или другой из его семейства):

```
PATH="$PATH:/home/jrh13/caml/bin"
export PATH
```

Чтобы не вводить эти команды при каждом входе на сервер, вы можете вставить их в конец вашего файла `.bash_profile` или его эквивалента для вашего командного процессора. Теперь, для использования CAML в интерактивном режиме, вам просто надо набрать `camllight`, и программа должна запуститься и выдать приглашение (`#`):

```
$ camllight
>      Caml Light version 0.73

#
```

Для того, чтобы выйти из системы, просто наберите `ctrl/d` или `quit();;` в строке ввода. Если вы заинтересованы в установке CAML Light на ваш собственный компьютер, то вы должны прочитать следующую Web-страницу для получения подробной информации:

```
http://pauillac.inria.fr/caml/
```

³Это имя означает ‘Categorical Abstract Machine’, метод реализации, лежащий в её основе.

5.5 Взаимодействие с ML

Когда ML выдаст вам строку приглашения, вы можете вводить выражения, завершённые двумя последовательными знаками “точка с запятой”. Принято говорить, что ML находится в режиме диалога (который также имеет жаргонное название REPL, read-eval-print loop): выражения считываются, вычисляются и выводятся результаты. Например, ML может быть использован как простой калькулятор:

```
#10 + 5;;
- : int = 15
```

Система не только возвращает ответ, но также выдаёт *тип* выражения, который определяется автоматически. Система может сделать это, поскольку знает тип встроенного оператора сложения `+`. С другой стороны, если для выражения не может быть определён тип, то система отвергнет его и постарается выдать сообщение о том, почему произошла ошибка. В сложных случаях сообщения об ошибках достаточно тяжело понять.

```
#1 + true;;
Toplevel input:
>let it = 1 + true;;
>
~~~~~
This expression has type bool,
but is used with type int.
```

Поскольку ML является функциональным языком, то выражения могут иметь функциональный тип. Для λ -абстракций $\lambda x. t[x]$ ML предоставляет следующий синтаксис — `fun x -> t[x]`. Например, мы можем определить функцию вычисления целого числа, следующего за данным:

```
#fun x -> x + 1;;
- : int -> int = <fun>
```

Как и в предыдущем примере, тип выражения (сейчас это `int -> int`), выводится и выдаётся на экран. Однако сама функция не печатается; система лишь выдаёт `<fun>`. Это сделано потому, что внутреннее представление функций не слишком читабельно.⁴ Функции применяются к следующим за ними аргументам, так же как и в λ -исчислении. Например:

```
 #(fun x -> x + 1) 4;;
- : int = 5
```

Подобно λ -исчислению, применение функции левоассоциативно, так что вы можете определять каррированные функции, используя то же самое соглашение по сокращению повторяющихся λ -абстракций (т.е., `fun-й`). Например, все следующие выражения эквивалентны:

```
#((fun x -> (fun y -> x + y)) 1) 2;;
- : int = 3
#(fun x -> fun y -> x + y) 1 2;;
- : int = 3
#(fun x y -> x + y) 1 2;;
- : int = 3
```

⁴CAML не хранит их как синтаксические деревья, а компилирует в байт-код.

5.6 Связывания и объявления

Вводить большое выражение целиком утомительно, гораздо удобнее будет воспользоваться `let` для связывания осмысленных подвыражений с именами. Это может быть сделано следующим образом:

```
#let successor = fun x -> x + 1 in
  successor(successor(successor 0));;
- : int = 3
```

Для связывания функций существует более элегантная конструкция:

```
#let successor x = x + 1 in
  successor(successor(successor 0));;
- : int = 3
```

в том числе для рекурсивных определений, дополненных ключевым словом `rec`:

```
#let rec fact n = if n = 0 then 1
                  else n * fact(n - 1) in
  fact 6;;
- : int = 720
```

Используя `and`, мы можем сделать несколько связываний одновременно и определить взаимно рекурсивные функции. Вот пример двух простых, хотя и сильно неэффективных, функций, которые определяют, является ли натуральное число чётным или нечётным:

```
#let rec even n = if n = 0 then true else odd (n - 1)
  and odd n = if n = 0 then false else even (n - 1);;
even : int -> bool = <fun>
odd : int -> bool = <fun>
#even 12;;
- : bool = true
#odd 14;;
- : bool = false
```

В действительности, любое связывание может быть сделано отдельно от его применения. ML помнит набор связанных переменных, и пользователь может пополнять данный набор интерактивно. Просто уберите `in` и завершите выражение двойной точкой с запятой:

```
#let successor = fun x -> x + 1;;
successor : int -> int = <fun>
```

После этого объявления, любое последующее выражение может использовать функцию `successor`, например:

```
#successor 11;;
- : int = 12
```

Заметьте, что мы не делаем *присваивания* значений *переменным*. Каждое связывание выполняется только раз, когда система анализирует введённые данные; оно не может быть повторено или изменено. Оно может быть перезаписано новым определением с тем же именем, но это не присваивание, в своём обычном значении, поскольку последовательность событий связана только с процессом *компиляции*, а не с динамикой *выполнения* программы. Конечно, отходя от интерактивного получения ответа от системы, мы можем полностью заменить все двойные точки с запятой, записанные после объявлений, на `in` и вычислить всё сразу. С этой точки зрения видно, что переписывание объявления в действительности соответствует определению новой локальной переменной, которая скрывает предыдущую, в соответствии с обычными правилами λ -исчисления. Например:

```
#let x = 1;;  
x : int = 1  
#let y = 2;;  
y : int = 2  
#let x = 3;;  
x : int = 3  
#x + y;;  
- : int = 5
```

является тем же самым, что и:

```
#let x = 1 in  
  let y = 2 in  
    let x = 3 in  
      x + y;;  
- : int = 5
```

Обязательно заметьте, что следуя λ -исчислению, связывание переменных является *статическим*, например, первое связывание `x` используется до того, как будет сделано новое связывание, и любое его использование до нового определения не будет затронуто этим определением. Например:

```
#let x = 1;;  
x : int = 1  
#let f w = w + x;;  
f : int -> int = <fun>  
#let x = 2;;  
x : int = 2  
#f 0;;  
- : int = 1
```

Первые версии LISP, однако, использовали *динамическое* связывание, когда переопределение переменной также распространялось на предыдущие использования этой переменной, так что аналогичная последовательность команд должна будет вернуть число 2. В действительности это считалось ошибкой, но скоро программисты начали использовать эту возможность. Как следствие, когда некоторая низкоуровневая функция изменялась, то изменения распространялись на все её применения в других функциях без необходимости перекомпиляции. Такая возможность продолжала существовать долгое время во многих диалектах LISP, но в конечном счёте победила идея, что статическое связывание лучше. В Common LISP по умолчанию используется статическое связывание, но динамическое также можно разрешить, если необходимо, используя ключевое слово `special`.

5.7 Полиморфные функции

Мы можем определять полиморфные функции, например, тождественное отображение:

```
#let I = fun x -> x;;
I : 'a -> 'a = <fun>
```

Внешнее представление в ASCII-кодировке типовых переменных α , β , ... в ML имеет вид 'a, 'b, ... Пример использования полиморфной функции с разными типами:

```
#I true;;
- : bool = true
#I 1;;
- : int = 1
#I I I I 12;;
- : int = 12
```

В данном примере все вхождения I имеют различные типы и интуитивно соответствуют разным функциям. Очередным этапом будет определение всех базовых комбинаторов:

```
#let I x = x;;
I : 'a -> 'a = <fun>
#let K x y = x;;
K : 'a -> 'b -> 'a = <fun>
#let S f g x = (f x) (g x);;
S : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

Заметьте, что система сама следит за типами, несмотря на то, что в последнем случае они были довольно сложными. Теперь, вспомним, что $I = S K K$; давайте попробуем сделать это на ML:⁵

```
#let I' = S K K;;
I' : '_a -> '_a = <fun>
```

Выражение имеет правильный тип⁶ и может быть легко проверено на конкретных случаях, например:

```
#I' 3 = 3;;
- : bool = true
```

В приведённых примерах полиморфных функций система очень быстро выводит наиболее общий тип для каждого выражения, и этот тип достаточно прост. Так обычно и происходит на практике, но существуют патологические случаи, например, следующий пример, приведённый в [Mairson \(1990\)](#). Тип этого выражения выводится около 10 секунд и занимает более 4000 строк на 80-символьном терминале.

⁵Отметим, что без учёта типов, $S K A = I$ верно для любого A . Однако, читатель может сам попробовать, например, $S K S$ и убедиться, что его тип является менее общим, чем ожидалось.

⁶Игнорируйте подчёркивания. Это связано с типизацией императивных конструкций, которые мы обсудим позже.

```

let pair x y = fun z -> z x y in
let x1 = fun y -> pair y y in
let x2 = fun y -> x1(x1 y) in
let x3 = fun y -> x2(x2 y) in
let x4 = fun y -> x3(x3 y) in
let x5 = fun y -> x4(x4 y) in
x5(fun z -> z);;

```

Мы уже упоминали, что программисты на ML никогда не задают типы. Это правда в том смысле, что транслятор ML сам назначит выражению наиболее общий тип. Однако, иногда бывает полезно *ограничить* вывод типа. Подобная мера не заставит работать код, который до этого не работал, но может использоваться как документация для понимания его предназначения; также возможно использовать более короткие синонимы для сложных типов. Ограничение типа может быть задано в ML путём добавления *аннотации типа* после некоторого выражения. Аннотации типов состоят из двоеточия, за которым указан тип. Обычно расположение аннотаций не имеет значения; если они есть, то они заставляют использовать соответствующие ограничения. Например, вот несколько альтернативных вариантов явного назначения тождественной функции типа `int -> int`:

```

#let I (x:int) = x;;
I : int -> int = <fun>
#let I x = (x:int);;
I : int -> int = <fun>
#let (I:int->int) = fun x -> x;;
I : int -> int = <fun>
#let I = fun (x:int) -> x;;
I : int -> int = <fun>
#let I = ((fun x -> x):int->int);;
I : int -> int = <fun>

```

5.8 Равенство функций

Вместо проверки эквивалентности функций, таких как I и I' , сравнением результатов их применения к конкретным аргументам, например 3, может показаться, что мы можем разрешить данный вопрос путём сравнения самих функций. Однако, это не работает:

```

#I' = I;;
Uncaught exception: Invalid_argument "equal: functional value"

```

В общем случае *запрещено* сравнивать функции, хотя в нескольких специальных случаях, когда равенство функций очевидно, выдаётся `true`.

```
#let f x = x + 1;;
f : int -> int = <fun>
#let g x = x + 1;;
g : int -> int = <fun>
#f = f;;
- : bool = true
#f = g;;
Uncaught exception: Invalid_argument "equal: functional value"
#let h = g;;
h : int -> int = <fun>
#h = f;;
Uncaught exception: Invalid_argument "equal: functional value"
#h = g;;
- : bool = true
```

Почему существуют эти ограничения? Разве в ML функции не равноправны с другими разновидностями данных? Да, но к сожалению, (экстенциональное) равенство функций в общем случае невычислимо. Это следует из различных классических результатов теории алгоритмов, таких как *неразрешимость проблемы останова* и *теорема Райса*.⁷ Приведём конкретный пример, демонстрирующий эту неразрешимость. На данный момент для функции, определённой ниже, так и не установлено, завершается ли её вычисление для произвольного аргумента. Предположение, что данная функция вычислима всюду, известно как *гипотеза Коллатца (Collatz conjecture)*.⁸

```
#let rec collatz n =
  if n <= 1 then 0
  else if even(n) then collatz(n / 2)
  else collatz(3 * n + 1);;
collatz : int -> int = <fun>
```

С другой стороны, очевидно, что в случае завершения вычислений их результатом всегда будет 0. Теперь рассмотрим следующую тривиальную функцию:

```
#let f (x:int) = 0;;
f : int -> int = <fun>
```

Решив уравнение `collatz = f`, компьютер подтвердил бы гипотезу Коллатца. Похожие примеры также легко построить на основе других математических задач, решение которых пока не найдено.

Процедура контроля типов может быть расширена таким образом, что позволит выявлять без выполнения программы попытки сравнения как элементарных объектов-функций, так и объектов-агрегатов, в состав которых входят объекты-функции. Типы данных, в которые не входят подвыражения типа функция, известны как *сравнимые типы*, поскольку всегда возможно проверить объекты таких типов

⁷Вы увидите доказательства в курсе теории алгоритмов. Теорема Райса — чрезвычайно сильный вывод о неразрешимости, которая утверждает, что *любое* нетривиальное свойство функции, соответствующее программе, невычислимо из её текста. Отличной книгой по теории алгоритмов является [Davis, Sigal, and Weyuker \(1994\)](#).

⁸Хороший обзор этой проблемы и попыток её решения приведены в [Lagarias \(1985\)](#). Строго говоря, мы должны использовать целые числа неограниченной разрядности, а не машинную арифметику. Позже мы увидим как это сделать.

на равенство. В то же время, это делает систему типов более сложной. Однако, сторонники такого подхода придерживаются мнения, что статическая проверка типов должна быть настолько полной, насколько это возможно.

Дополнительная литература

Многие книги о функциональном программировании содержат информацию по общим для всех функциональных языков аспектам, которые мы тоже обсуждали, например, по стратегии вычислений. Хорошее элементарное введение в CAML Light и функциональное программирование можно найти в [Mauney \(1995\)](#). [Paulson \(1991\)](#) также является хорошей книгой, хотя основывается на Standard ML.

Примеры

1. Предположим, что ‘функция-условие’, определённая $\text{ite}(b, x, y) = \text{if } b \text{ then } x \text{ else } y$ является единственной функцией, которая может работать с аргументами типа `bool`. Существует ли способ написать функцию факториала?
2. Докажите при помощи правил типизации, приведённых в предыдущей главе, что комбинатор S имеет в точности тот тип, который выводит для него ML.
3. Напишите простую рекурсивную функцию возведения в степень целых чисел, т.е. вычисляющую x^n для $n \geq 0$. Напишите на ML пару функций, из эквивалентности которых следовала бы справедливость Великой теоремы Ферма: не существует целых чисел x, y, z и натурального числа $n > 2$ таких, что $x^n + y^n = z^n$, за исключением тривиального случая, когда $x = 0$ или $y = 0$.

Глава 6

Более подробно о ML

В этой главе мы закрепим предыдущие примеры, более точно определив синтаксис и основные возможности ML, а затем перейдём к рассмотрению некоторых дополнительных возможностей, таких как рекурсивные типы. Но начнём обсуждения взаимодействия с системой.

До сих пор мы печатали команды в интерпретатор, получая от него результат. Однако, это не лучший способ писать нетривиальные программы. Обычно вам следует записывать все выражения и объявления в файл. После этого повторное использование кода обеспечивается операцией «вырезать и вставить». Эта операция может быть выполнена с помощью X Window System и схожих систем, или в редакторе типа Emacs. Такой подход, однако, становится утомительным и неэффективным для больших программ. Вместо этого вы можете использовать функцию `include` для непосредственного чтения из файла. Например, если файл `myprog.ml` содержит:

```
let pythag x y z =  
  x * x + y * y = z * z;;  
  
pythag 3 4 5;;  
  
pythag 5 12 13;;  
  
pythag 1 2 3;;
```

то введя в интерпретатор `include "myprog.ml";;` вы получите:

```
#include "myprog.ml";;  
pythag : int -> int -> int -> bool = <fun>  
- : bool = true  
- : bool = true  
- : bool = false  
- : unit = ()
```

То есть, вывод системы ML получился такой же, как если бы вы печатали содержимое файла в консоль. Последняя строка вывода – это результат вычисления самого выражения `include`

В больших программах часто полезно включать комментарии. В ML они записываются между символами `(*` и `*)`, например:

```

(* _____ *)
(* This function tests if (x,y,z) is a Pythagorean triple *)
(* _____ *)

let pythag x y z =
  x * x + y * y = z * z;;

(*comments*) pythag (*can*) 3 (*go*) 4 (*almost*) 5 (*anywhere*)
(* and (* can (* be (* nested *) quite *) arbitrarily *) *);;

```

6.1 Основные типы данных и операции

ML представляет несколько встроенных примитивных типов. Из них, с помощью конструкторов типов, могут быть построены составные типы. Пока мы будем использовать только конструктор функциональных типов `->` и конструктор декартового произведения типов `*`. В своё время мы рассмотрим и другие конструкторы, а также узнаем как определять новые типы и конструкторы типов. Примитивные типы, которые интересуют нас, это:

- Тип `unit`, можно называть его процедурным. Это одноэлементный тип, единственный элемент которого записывается как `()`. Очевидно, использование типа `unit` не передаёт какой-либо информации, так что он часто используется как возвращаемый тип императивно записанных «функций», результат которых не значение вычислений, а какой-либо побочный эффект. Например, таких как `include`. Он так же может использоваться как тип функции, используемой для приостановки вычислений.
- Тип `bool`. Двухэлементный тип булевых значений (значений истинности). Элементы этого типа: `true` и `false`.
- Тип `int`. Содержит конечное подмножество отрицательных и неотрицательных чисел. Обычно значения варьируются от -2^{30} (-1073741824) до $2^{30} - 1$ (1073741823).¹ Элементы записываются обычным способом, например: `0`, `32`, `-25`.
- Тип `string` содержит строки (т.е. конечные последовательности) символов. Они записываются и печатаются в двойных кавычках, вот так: `"hello"`. Для записи специальных символов используется экранирование в стиле языка C. Например, `"` - запись двойной кавычки, `\n` - символ перевода строки.

Значения, такие как `()`, `false`, `7` и `"saml"`, в смысле лямбда - исчисления рассматриваются как константы. Другие константы соответствуют *операциям* на основных типах. Некоторые из них, из соображений привычности, могут быть записаны как инфиксные. Для операций определено понятие приоритета,

¹На самом деле, в машинной арифметике значения ещё более ограничены, поскольку один бит используется сборщиком мусора (см. определения). Далее мы увидим, как использовать альтернативный тип целых с неограниченной точностью.

поэтому выражения группируются так, как ожидается. Например, мы пишем $x + y$ вместо $+ x y$, и $x < 2 * y + z$ вместо $< x (+ (* 2 y) z)$. Логический оператор `not` выделяется тем, что имеет особое правило разбора, для него не выполняется обычное правило ассоциативности слева: `not not p` означает `not (not p)`. Функцию, определённую пользователем, можно задать как инфиксную, с помощью директивы `#infix`. Например, вот определение функции, выполняющей композицию функций:

```
#let successor x = x + 1;;
successor : int -> int = <fun>
#let o f g = fun x -> f(g x);;
o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
#let add3 = o successor (o successor successor);;
add3 : int -> int = <fun>
#add3 0;;
- : int = 3
##infix "o";;
#let add3' = successor o successor o successor;;
add3' : int -> int = <fun>
#add3' 0;;
- : int = 3
```

В ML нет возможностей задавать приоритет для инфиксных операций, определённых пользователем, а также определять неинфиксные операции как правоассоциативные. Также запомните, что неявная операция «апликация функции» имеет приоритет выше, чем у любой другой бинарной операции, так что `successor 1 * 2` разбирается как `(successor 1) * 2`. Если хотите использовать функцию с особым статусом как обычную константу, вам потребуется `prefix`:

```
#o successor successor;;
Toplevel input:
>o successor successor;;
>^
Syntax error.
#prefix o successor successor;;
- : int -> int = <fun>
#(prefix o) successor successor;;
- : int -> int = <fun>
```

Разобравшись с этими деталями синтаксиса, давайте рассмотрим список операций на основных типах. Унарные операции:

Оператор	Тип	Значение
<code>-</code>	<code>int -> int</code>	Численное отрицание
<code>not</code>	<code>bool -> bool</code>	Логическое отрицание

и бинарные операции, в порядке убывания приоритета:

Оператор	Тип	Значение
<code>mod</code>	<code>int -> int -> int</code>	Взятие модуля
<code>*</code>	<code>int -> int -> int</code>	Произведение
<code>/</code>	<code>int -> int -> int</code>	Целочисленное деление
<code>+</code>	<code>int -> int -> int</code>	Сложение
<code>-</code>	<code>int -> int -> int</code>	Вычитание
<code>^</code>	<code>string -> string -> string</code>	Склейка строк
<code>=</code>	<code>'a -> 'a -> bool</code>	Равенство
<code><></code>	<code>'a -> 'a -> bool</code>	Неравенство
<code><</code>	<code>'a -> 'a -> bool</code>	Меньше чем
<code><=</code>	<code>'a -> 'a -> bool</code>	Меньше или равно
<code>></code>	<code>'a -> 'a -> bool</code>	Больше чем
<code>>=</code>	<code>'a -> 'a -> bool</code>	Больше или равно
<code>&</code>	<code>bool -> bool -> bool</code>	Логическое «и»
<code>or</code>	<code>bool -> bool -> bool</code>	Логическое «или»

Например, `x > 0 & x < 1` парсится как `& (> x 0) (< x 1)`. Заметьте, что все сравнения, не только по равенству, полиморфны. Они не только соответствующим образом упорядочивают числа и строчки, но также и остальные примитивные, и составные типы. Однако, они не применимы к функциям.

Две логические операции `&` и `or` имеют особую стратегию вычисления. В сущности, их можно рассматривать как синонимы условных выражений:

$$\begin{aligned}
 p \ \& \ q &\triangleq \text{if } p \text{ then } q \text{ else false} \\
 p \ \text{or} \ q &\triangleq \text{if } p \text{ then true else } q
 \end{aligned}$$

Следовательно, операция «and» вычисляет первый аргумент, и только если его значение true, вычисляет следующий. Напротив, «or» вычисляет первый аргумент, и только если это false, вычисляет второй.

Выражения в ML строятся из констант и переменных; любой идентификатор, не связанный на текущий момент выполнения программы, рассматривается как переменная. Объявления связывают значения выражений с именами, при этом в состав выражений могут входить другие объявления. Следовательно, синтаксические классы выражений и объявлений взаимно рекурсивны. Мы можем выразить это следующей БНФ - грамматикой.²

$$\begin{aligned}
 \textit{expression} ::= & \textit{variable} \\
 & | \textit{constant} \\
 & | \textit{expression expression} \\
 & | \textit{expression infix expression} \\
 & | \textit{not expression} \\
 & | \textit{if expression then expression else expression} \\
 & | \textit{fun pattern} \rightarrow \textit{expression}
 \end{aligned}$$

²Мы отбросили многие конструкции, которые нами не будут использоваться. За подробностями обращайтесь к руководству по CAML.

```

| (expression)
| declaration in expression
declaration ::= let let_bindings
| let rec let_bindings
let_bindings ::= let_binding
| let_binding and let_bindings
let_binding ::= pattern = expression
pattern ::= variables
variables ::= variable
| variable variables

```

Позже мы более подробно обсудим синтаксический класс *pattern*. До сих пор мы имели дело только с *variable* или *variable variable ... variable*. В первом случае мы просто связываем выражение с именем, тогда как во втором, для объявления функций, используется специальный синтаксический сахар. Аргументы функции записываются после имени функции, слева от знака равенства. Например, вот объявление функции, `add4`, которая может использоваться для прибавления 4 к её аргументу:

```

#let add4 x =
  let y = successor x in
  let z = let w = successor y in
    successor w in
  successor z;;
add4 : int -> int = <fun>
#add4 1;;
- : int = 5

```

Представляется полезным разбор этого объявления в соответствии с приведённой выше грамматикой.

6.2 Дальнейшие примеры

Не сложно задать рекурсивную функцию, принимающую в качестве аргументов целочисленное n и функцию f , а возвращающую f^n , т.е. $f \circ \dots \circ f$ (n раз):

```

#let rec funpow n f x =
  if n = 0 then x
  else funpow (n - 1) f (f x);;
funpow : int -> ('a -> 'a) -> 'a -> 'a = <fun>

```

В действительности, небольшое размышление показывает, что функция `funpow` принимает машинное целое n и возвращает нумерал (число) Чёрча, соответствующее n . Поскольку функции не выводятся на печать, мы не можем в действительности посмотреть на лямбда-выражение, представляющее число Чёрча:

```

#funpow 6;;
- : ('_a -> '_a) -> '_a -> '_a = <fun>

```

Также просто задаётся функция обратная к `funpow`, которая принимает число Чёрча и возвращает машинное целое:

```
#let defrock n = n (fun x -> x + 1) 0;;
defrock : ((int -> int) -> int -> 'a) -> 'a = <fun>
#defrock(funpow 32);;
- : int = 32
```

Мы можем проверить некоторые из арифметических операций на числах Чёрча:

```
#let add m n f x = m f (n f x);;
add : ('a -> 'b -> 'c) -> ('a -> 'd -> 'b) -> 'a -> 'd -> 'c = <fun>
#let mul m n f x = m (n f) x;;
mul : ('a -> 'b -> 'c) -> ('d -> 'a) -> 'd -> 'b -> 'c = <fun>
#let exp m n f x = n m f x;;
exp : 'a -> ('a -> 'b -> 'c -> 'd) -> 'b -> 'c -> 'd = <fun>
#let test bop x y = defrock (bop (funpow x) (funpow y));;
test :
  (((('a -> 'a) -> 'a -> 'a) ->
    (('b -> 'b) -> 'b -> 'b) -> (int -> int) -> int -> 'c) ->
    int -> int -> 'c = <fun>
#test add 2 10;;
- : int = 12
#test mul 2 10;;
- : int = 20
#test exp 2 10;;
- : int = 1024
```

Примеры выше, очевидно, не являются самым эффективным способом выполнения арифметических операций. Далее, ML не имеет функции для возведения в степень, но её легко задать по рекурсии:

```
#let rec exp x n =
  if n = 0 then 1
  else x * exp x (n - 1);;
exp : int -> int -> int = <fun>
```

Это решение выполняет n произведений для вычисления x^n . Более эффективный способ использует тот факт, что $x^{2n} = (x^n)^2$ и $x^{2n+1} = x(x^n)^2$:

```
#let square x = x * x;;
square : int -> int = <fun>
#let rec exp x n =
  if n = 0 then 1
  else if n mod 2 = 0 then square(exp x (n / 2))
  else x * square(exp x (n / 2));;
exp : int -> int -> int = <fun>
#infix "exp";;
#2 exp 10;;
- : int = 1024
#2 exp 20;;
- : int = 1048576
```

Другая классическая операция на натуральных числах - нахождение их наибольшего общего делителя при помощи алгоритма Евклида:

```
#let rec gcd x y =
  if y = 0 then x else gcd y (x mod y);;
gcd : int -> int -> int = <fun>
#gcd 100 52;;
- : int = 4
#gcd 7 159;;
- : int = 1
#gcd 24 60;;
- : int = 12
```

Мы использовали оператор рекурсии `Rec` при объяснении рекурсивных определений. Действительно, давайте определим его:

```
#let rec Rec f = f(fun x -> Rec f x);;
Rec : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
#let fact = Rec (fun f n -> if n = 0 then 1 else n * f(n - 1));;
fact : int -> int = <fun>
#fact 3;;
it : int = 6
```

Обратите внимание на необходимость анонимной функции, в противном случае выражение `Rec f` уходит в бесконечную последовательность рекурсивных вызовов, прежде чем функция будет применена к своему аргументу:

```
#let rec Rec f = f(Rec f);;
Rec : ('a -> 'a) -> 'a = <fun>
#let fact = Rec (fun f n -> if n = 0 then 1 else n * f(n - 1));;
Uncaught exception: Out_of_memory
```

6.3 Определения типов

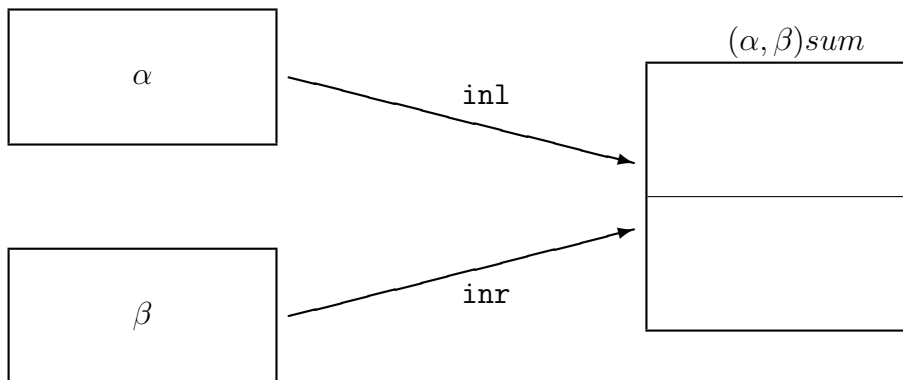
Мы уже говорили о том, что ML имеет возможности для объявления новых конструкторов типов, так что сложные типы данных могут быть построены на базе существующих. В действительности, ML позволяет создавать типы не только из существующих типов, но и из самих сложных типов данных. Такие типы называются *рекурсивными*. Они объявляются с помощью ключевого слова `type`, за которым выражение, описывающее как тип создаётся из существующих типов и себя. Мы покажем это на нескольких примерах. Первым примером будет определение типа *sum*, которые должен соответствовать несвязному объединению (\sqcup) двух существующих типов.

```
#type ('a,'b)sum = inl of 'a | inr of 'b;;
Type sum defined.
```

Грубо говоря, объект типа `('a,'b)sum` является *либо* чем-то типа `'a`, *либо* чем-то типа `'b`. Однако, говоря более формально, все эти объекты имеют разные типы. Объявление типа, также определяет так называемые *конструкторы* `inl` и `inr`. Это функции, которые получают объекты с типами компонентов, и встраивают их в новый тип. Мы действительно можем видеть их типы в ML и применять их к объектам:

```
#inl;;
- : 'a -> ('a, 'b) sum = <fun>
#inr;;
- : 'a -> ('b, 'a) sum = <fun>
#inl 5;;
- : (int, 'a) sum = inl 5
#inr false;;
- : ('a, bool) sum = inr false
```

Мы можем показать данную ситуацию с помощью следующей диаграммы. При наличии двух существующих типов α и β , тип $(\alpha, \beta)sum$ создаётся из точных отдельных копий α и β , и два конструктора отображаются на соответствующие копии:



Это аналогично использованию `union` в языке C, но в ML копии типов компонентов хранятся отдельно и кто-то всегда знает, к какому из этих элементов относится объединение. В противоположность этому, в C типы компонентов пересекаются и программист отвечает за корректность работы с ними.

6.3.1 Сопоставление с образцом

Конструкторы в таком определении имеют три очень важных качества:

- Они полные (exhaustive), т.е., каждый элемент нового типа может быть получен через `inl x` для некоторого `x` или через `inr y` для некоторого `y`. Так что, новый тип не содержит ничего, кроме копий типов компонент.
- Они являются инъективными, т.е. тест на равенство `inl x = inl y` даёт истинный результат, если, и только если `x = y`, и аналогичным образом, для `inr`. Так что, новый тип содержит точную копию типа каждого компонента, без идентификации любого из элементов.
- Они индивидуальны, т.е. их диапазоны несвязны. Это означает, что в предыдущем примере `inl x = inr y` будет ложным, независимо от того, чем являются `x` и `y`. Так что, копия типа каждого компонента в новом типе, хранится независимо.

Второе и третье свойство конструкторов выравнивает (justify) наше использование *сопоставления с образцом*. Это выполняется за счёт использования более общих *varstructs* как аргументов в *lambda*-выражениях, например:


```
#fun (inl n) -> n > 6
  | (inr b) -> b;;
- : (int, bool) sum -> bool = <fun>
```

Эта функция имеет свойство (достаточно натуральное), что когда она применяется к `inl n` она возвращает `n > 6`, а когда она применяется к `inr b`, то она возвращает `b`. Это происходит в соответствии со вторым и третьим свойствами конструкторов, что даёт нам хорошо определённую функцию. Поскольку конструкторы являются инъективными, мы можем точно восстановить `n` из `inl n` и `b` из `inr b`. Поскольку конструкторы индивидуальны, мы знаем, что два выражения не могут быть взаимно неконсистентны, поскольку никакое значение не соответствует обоим образцам.

В добавление к этому, поскольку конструкторы являются полными (??? exhaustive), мы знаем, что каждое значение будет подпадать под один из образцов, так что функция определена везде. В действительности, разрешается ослабить это свойство путём неуказания некоторых образцов, хотя при этом ML выдаст предупреждение:

```
#fun (inr b) -> b;;
Toplevel input:
>fun (inr b) -> b;;
>~~~~~
Warning: this matching is not exhaustive.
- : ('a, 'b) sum -> 'b = <fun>
```

Если эта функция применяется к чему-то вида `inl x`, то она не будет работать:

```
#let f = fun (inr b) -> b;;
Toplevel input:
>let f = fun (inr b) -> b;;
>~~~~~
Warning: this matching is not exhaustive.
f : ('a, 'b) sum -> 'b = <fun>
#f (inl 3);;
Uncaught exception: Match_failure ("", 452, 468)
```

Хотя логические значения и встроены в ML, они эффективно определяются тривиальным использованием рекурсивного типа, часто называемого (*enumeration*) *enumerated type*, где конструкторы не получают аргументов:

```
#type bool = false | true;;
```

В самом деле, разрешается определять выражения путём сопоставления с истинным значением. Следующие два выражения полностью эквивалентны:

```
#if 4 < 3 then 1 else 0;;
- : int = 0
#(fun true -> 1 | false -> 0) (4 < 3);;
- : int = 0
```

Однако, сопоставление с образцом не ограничивается определением элементов рекурсивных типов, хотя в частности это очень удобно. Например, мы можем определить функцию, которая сообщает нам является ли число нулём, следующим образом:

```
#fun 0 -> true | n -> false;;
- : int -> bool = <fun>
#(fun 0 -> true | n -> false) 0;;
- : bool = true
#(fun 0 -> true | n -> false) 1;;
- : bool = false
```

В этом случае, мы не имеем взаимной исключительности образцов, поскольку 0 соответствует обоим образцам. Образцы сопоставляются по порядку, один за другим, и используется первый соответствующий. Заметьте, что пока соответствия не являются взаимно исключающими, нет гарантии, что каждое выражение содержит математическое уравнение. Например, в предыдущем примере, функция не возвращает `false` для любого `n`, так что второе выражение не универсально.

Заметьте, что только *конструкторы* могут использоваться в вышеприведённых выражениях как компоненты образцов. Обычные константы будут рассматриваться как новые переменные, связанные внутри образца. Например, рассмотрим следующий код:

```
#let true_1 = true;;
true_1 : bool = true
#let false_1 = false;;
false_1 : bool = false
#(fun true_1 -> 1 | false_1 -> 0) (4 < 3);;
Toplevel input:
>(fun true_1 -> 1 | false_1 -> 0) (4 < 3);;
>
Warning: this matching case is unused.
- : int = 1
```

В общем, элемент-модуль (unit element) `()`, истинные значения, целые числа, строковые константы и операция образования пар (infix comma) имеют статус конструкторов, также как и другие конструктора из предопределённых рекурсивных типов. Когда они встречаются в образце, соответствующее значение должно иметь соответствующий тип. Все другие идентификаторы соответствуют любому выражению и становятся связанными в процессе обработки.

Также как и `???varstructs` в λ -выражениях, существуют другие способы выполнения сопоставления с образцом. Вместо создания функции с помощью сопоставления с образцом, и применения её к выражению, мы можем выполнять сопоставление с образцом напрямую используя следующую конструкцию:

$$\text{match } expression \text{ with } pattern_1 \rightarrow E_1 \mid \dots \mid pattern_n \rightarrow E_n$$

Простейшей альтернативой этому является использование

$$\text{let } pattern = expression$$

но в этом случае разрешено использование лишь одного образца.

6.3.2 Рекурсивные типы

Предыдущие примеры были бессодержательны в том смысле, что мы не определяли тип через самое себя. В качестве более интересного примера давайте объявим тип списков ³ элементов типа 'a.

```
#type ('a)list = Nil | Cons of 'a * ('a)list;;
Type list defined.
```

Проверим типы конструкторов:

```
#Nil;;
- : 'a list = Nil
#Cons;;
- : 'a * 'a list -> 'a list = <fun>
```

Конструктор Nil, не принимающий никаких аргументов, просто создаёт некоторый объект типа ('a)list, который рассматривается как пустой список. Другой конструктор Cons принимает аргументы типа 'a и ('a)list, а возвращает новый список, который образован из предыдущего добавлением в начало первого элемента. Например:

```
#Nil;;
- : 'a list = Nil
#Cons(1,Nil);;
- : int list = Cons (1, Nil)
#Cons(1,Cons(2,Nil));;
- : int list = Cons (1, Cons (2, Nil))
#Cons(1,Cons(2,Cons(3,Nil)));;
- : int list = Cons (1, Cons (2, Cons (3, Nil)))
```

Поскольку значение конструктора инъективно зависит от аргументов, легко понять, что все значения, которые мы рассматривали как списки [], [1], [1;2] и [1;2;3], различны, а списки различной длины будут иметь различный тип. Фактически, ML уже имеет тип list, точно такой, как только что был определён. Единственная разница в синтаксисе: пустой список записывается как [], а рекурсивный конструктор :: – инфиксный. Следовательно, все упомянутые списки в действительности записываются:

```
#[];;
- : 'a list = []
#1::[];;
- : int list = [1]
#1::2::[];;
- : int list = [1; 2]
#1::2::3::[];;
- : int list = [1; 2; 3]
```

Списки выводятся в удобном виде, это же справедливо и для ввода. Тем не менее, следует помнить, что это синтаксический сахар, а списки можно выражать и явно, в терминах конструкторов. Например, мы можем, используя сравнение по образцу, определить функции, принимающие начало и хвост списка.

³конечных упорядоченных последовательностей

```
#let hd (h::t) = h;;
Toplevel input:
>let hd (h::t) = h;;
>
Warning: this matching is not exhaustive.
hd : 'a list -> 'a = <fun>
#let tl (h::t) = t;;
Toplevel input:
>let tl (h::t) = t;;
>
Warning: this matching is not exhaustive.
tl : 'a list -> 'a list = <fun>
```

Компилятор предупреждает нас, что применение этих функций к пустым спискам приведёт к ошибкам. Посмотрим на них в действии:

```
#hd [1;2;3];;
- : int = 1
#tl [1;2;3];;
- : int list = [2; 3]
#hd [];;
Uncaught exception: Match_failure
```

Обратите внимание, следующее определение `hd` не является корректным. В действительности, функция принимает в качестве аргумента список ровно из двух аргументов, а на списках отличной длины не определена:

```
#let hd [x;y] = x;;
Toplevel input:
>let hd [x;y] = x;;
>
Warning: this matching is not exhaustive.
hd : 'a list -> 'a = <fun>
#hd [5;6];;
- : int = 5
#hd [5;6;7];;
Uncaught exception: Match_failure
```

Сравнение по образцу может комбинироваться с рекурсией. Например, вот функция, возвращающая длину списка:

```
#let rec length =
  fun [] -> 0
    | (h::t) -> 1 + length t;;
length : 'a list -> int = <fun>
#length [];;
- : int = 0
#length [5;3;1];;
- : int = 3
```

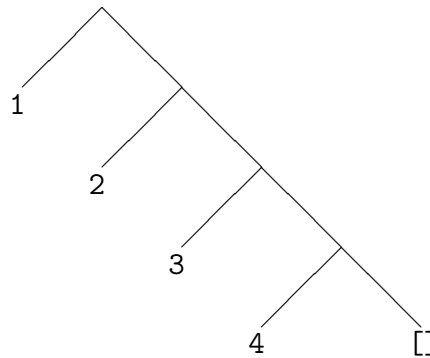
В качестве альтернативы можно задать её с помощью функций `hd` и `tl`:

```
#let rec length l =
  if l = [] then 0
  else 1 + length(tl l);;
```

Последний стиль задания функции весьма распространён во многих языках, особенно в LISP, но непосредственное сравнение по образцу часто бывает более элегантно.

6.3.3 Древовидные структуры

Часто бывает полезно представлять части рекурсивных типов в виде дерева с рекурсивными конструкторами в узлах, и остальными типами данных в качестве листьев. Рекурсивность просто говорит о том как дерево образуется из отдельных поддеревьев. В случае списков «деревья» более вытянутые и односторонние. Так, список `[1;2;3;4]` представляется в следующем виде:



Рекурсивные типы, допускающие построение более сбалансированных деревьев, задаются также просто, например:

```
#type ('a)btree = Leaf of 'a
                | Branch of ('a)btree * ('a)btree;;
```

В общем, может быть несколько различных рекурсивных конструкторов, с различным числом потомков. Это даёт очень простой способ представления *синтаксических деревьев* для языков программирования (и других формальных языков). Например, вот тип для представления арифметических выражений, построенных из целых чисел, операций сложения и умножения.

```
#type expression = Integer of int
                  | Sum of expression * expression
                  | Product of expression * expression;;
```

А вот рекурсивная функция для вычисления таких выражений:

```
#let rec eval =
  fun (Integer i) -> i
    | (Sum(e1,e2)) -> eval e1 + eval e2
    | (Product(e1,e2)) -> eval e1 * eval e2;;
eval : expression -> int = <fun>
#eval (Product(Sum(Integer 1,Integer 2),Integer 5));;
- : int = 15
```

Подобные абстрактные синтаксические деревья могут быть весьма успешным представлением данных, допускающим всевозможные манипуляции над ними. Так, часто первое действие компиляторов заключается в переводе, согласно правилам синтаксического разбора, текста на входе в «абстрактное синтаксическое дерево». Обратите внимание, что при этом условности, вроде приоритета или использования скобок, уже не требуются. Как только мы достигаем уровня абстрактного синтаксиса,

всё выражается явно за счёт структуры дерева. Мы будем использовать подобную технику при написании ML версий формальных правил лямбда-исчисления. Начнём с типа, используемого для представления лямбда-термов.

```
#type term = Var of string
           | Const of string
           | Comb of term * term
           | Abs of string * term;;
Type term defined.
```

Обратите внимание, вместо двух термов Abs принимает имя переменной и терм. Это сделано для запрещения некорректных термов. В качестве примера представим терм $\lambda x y. y (x x y)$ с помощью:

```
Abs("x",Abs("y",Comb(Var "y",Comb(Comb(Var "x",Var "x"),Var "y"))))
```

Вот определение рекурсивной функции `free_in`, определяющей является ли переменная в терме свободной:

```
#let rec free_in x =
  fun (Var v) -> x = v
    | (Const c) -> false
    | (Comb(s,t)) -> free_in x s or free_in x t
    | (Abs(v,t)) -> not x = v & free_in x t;;
free_in : string -> term -> bool = <fun>
#free_in "x" (Comb(Var "f",Var "x"));;
- : bool = true
#free_in "x" (Abs("x",Comb(Var "x",Var "y")));;
- : bool = false
```

Точно так мы можем по рекурсии определить подстановку. Сначала, во избежание конфликта имён, нам потребуется функция для переименования переменных. Мы хотим преобразовывать существующие имена переменных в новые, связанные в заданном выражении. Переименование производится путём добавления символа.

```
#let rec variant x t =
  if free_in x t then variant (x^'') t
  else x;;
variant : string -> term -> string = <fun>
#variant "x" (Comb(Var "f",Var "x"));;
- : string = "x'"
#variant "x" (Abs("x",Comb(Var "x",Var "y")));;
- : string = "x"
#variant "x" (Comb(Var "f",Comb(Var "x",Var "x'")));;
- : string = "x''"
```

Теперь мы можем определить подстановку так, как мы это сделали для абстрактных термов:

```
#let rec subst u (t,x) =
  match u with
  | Var y -> if x = y then t else Var y
  | Const c -> Const c
  | Comb(s1,s2) -> Comb(subst s1 (t,x),subst s2 (t,x))
  | Abs(y,s) -> if x = y then Abs(y,s)
                 else if free_in x s & free_in y t then
                     let z = variant y (Comb(s,t)) in
                     Abs(z,subst (subst s (Var y,z)) (t,x))
                 else Abs(y,subst s (t,x));;
subst : term -> term * string -> term = <fun>
```

Обратите внимание на очень близкие параллели между стандартным математическим представлением лямбда - исчисления, что обсуждалось ранее, и ML версией. Всё что нам действительно надо для завершения аналогии, это вместо явного вызова конструкторов читать и записывать термы в более удобной форме. Мы вернёмся к этой проблеме позже и обсудим как можно добавить такие возможности.

6.3.4 Тонкости рекурсивных типов

Рекурсивный тип может содержать вложенные конструкторы других типов, включая конструктор типа функций. Например, рассмотрим следующий код:

```
#type ('a)embedding = K of ('a)embedding->'a;;
Type embedding defined.
```

Если мы задумаемся о нижележащей семантике, то увидим, ситуация представляется серьёзной. Рассмотрим, к примеру, специальный случай, когда `'a` является `bool`. У нас также есть инъективная функция $K:((\text{bool})\text{embedding} \rightarrow \text{bool}) \rightarrow (\text{bool})\text{embedding}$. А это прямо противоречит теореме Кантора о том, что множество всех подмножеств множества X не может быть инъективно ему.⁴ Следовательно, рассуждая о типах, нам следует быть более осторожными. В действительности, $\alpha \rightarrow \beta$ не может рассматриваться как полное функциональное пространство. Иначе конструкции рекурсивных типов будут противоречивы. Поскольку все функции, которые мы можем задать, в действительности являются вычислимыми, есть смысл ограничиться только ими. С таким ограничением семантика будет непротиворечива, хотя детали усложнятся.

Рассмотренное определение также имеет интересные последствия для системы типов. Например, мы можем теперь определить Y комбинатор, используя K для приведения типа. Заметьте, если K удалены, мы имеем, в сущности, обычное определение Y комбинатора в нетипизированном лямбда - исчислении. Конструкция `let` используется всего лишь для эффективности, но мы *нуждаемся* в η -редексе включающем z для предупреждения заикливания в стратегии вычисления принятой в ML.

⁴Доказательство: рассмотрим $C = \{i(s) \mid s \in \wp(X) \text{ и } i(s) \notin s\}$. Если $i: \wp(X) \rightarrow X$ инъективно, выполняется $i(C) \in C \Leftrightarrow i(C) \notin C$, получили противоречие. Эта конструкция напоминает парадокс Рассела, и, возможно, в действительности вдохновлена им. Аналогия будет ещё более близкой, если мы рассмотрим равнозначное утверждение о несюръективности $j: X \rightarrow \wp(X)$, и докажем это, рассматривая $\{s \mid s \notin j(s)\}$.

```
#let Y h =
  let g (K x) z = h (x (K x)) z in
  g (K g);;
Y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
#let fact = Y (fun f n -> if n = 0 then 1 else n * f(n - 1));;
fact : int -> int = <fun>
#fact 6;;
- : int = 720
```

Итак, рекурсивные типы представляют собой значительный вклад в язык и позволяют нам обойтись без задания оператора рекурсии примитивной языка.

Упражнения

1. Что будет неверно в нашем определении `subst`, если мы заменим `Var y -> if x = y then t else Var y` следующим фрагментом? `Var x -> t | Var y -> Var y`
2. Можно было бы, определить (хотя это неэффективно) натуральные числа посредством следующего рекурсивного типа:

```
#type num = Zero | Suc of num;;
```

Напишите функции, выполняющие арифметические действия над числами в этой форме.

3. Используйте определение типа для синтаксиса лямбда - термов, что мы дали выше. Напишите функцию, конвертирующую произвольный лямбда - терм в эквивалент, записанный с помощью *S*, *K* и *I* комбинаторов. Представьте комбинаторы с помощью `Const "S"`, используйте технику, рассмотренную нами ранее.
4. Расширьте синтаксис лямбда - термов, включив типизацию в стиле Чёрча. Write functions that check the typing conditions before allowing terms to be formed, but otherwise map down to the primitive constructors.
5. Рассмотрим тип бинарных деревьев со строками в узлах:

```
#type stree = Leaf | Branch of stree * string * stree;;
```

Напишите функцию `strings`, возвращающую список строк, полученный при обходе дерева в порядке слева на право. Напишите функцию, которая принимает строку и дерево, для которого соответствующий список сортирован в алфавитном порядке, а возвращает дерево, со строкой, вставленной с сохранением упорядоченности.

6. (*) Улучшите эти функции так, чтобы на каждом шагу дерево оставалось почти балансированным, т.е. для каждого узла максимальная глубина левого и правого поддерева отличалась не более чем на единицу.⁵
7. (*) Можете ли вы описать типы выражений, которые могут быть записаны в типизированном лямбда - исчислении, без оператора рекурсии?⁶ Каков тип следующей рекурсивной функции?

```
let rec f i = (i o f) (i o i);;
```

Покажите как это может использоваться для записи ML выражений с полностью полиморфным типом α . Можете ли вы написать завершённое выражение с таким свойством? Как насчёт произвольной функции типа $\alpha \rightarrow \beta$?

⁵За подробностями об этих «АВЛ деревьях» и похожих методах для балансированных деревьев обращайтесь к [Adel'son-Vel'skii and Landis \(1962\)](#) и различным книгам по алгоритмам.

⁶Ответ лежит в «гомоморфизме Карри-Говарда» – для подробностей см. [Girard, Lafont, and Taylor \(1989\)](#).

Глава 7

Доказательство корректности программ

Программисты не раз убеждались на своём горьком опыте как бывает сложно написать *корректную* программу, то есть такую, которая делает в точности то, что требуется. В большинстве объёмных программ есть ошибки, последствия которых могут быть самыми различными. Некоторые ошибки безобидны, другие — раздражают, а некоторые — смертельно опасны. К примеру, программное обеспечение электрокардиостимуляторов, автопилотов, систем управления двигателями, антиблокировочных тормозных систем, приборов радиационной терапии, систем управления ядерных реакторов *критично к наличию ошибок*. Ошибки, допущенные при разработке таких программ, могут повлечь за собой массовые человеческие жертвы. Чем глубже проникновение компьютеров во все сферы жизни человечества, тем серьёзнее возможная угроза жизни людей, порождаемая ошибками в программах.¹

Каким образом мы можем убедиться в корректной работе программы? Одним из полезных методов будет тестирование программы на обширном наборе планируемых сценариев её использования. При этом входные данные для тестов подбираются так, чтобы проверить различные составляющие программы в поисках возможных ошибок. В реальности оказывается, что потенциальных вариантов использования слишком много, чтобы полностью проверить все из них, так что некоторые ошибки могут быть пропущены в ходе тестирования. Как неоднократно подчёркивают многие авторы, в частности, Дейкстра, тестирование программ может быть полезным для демонстрации наличия ошибок, но показать их отсутствие оно в состоянии лишь в отдельных редких случаях.

Альтернативой тестированию программ служит возможность их математической *верификации*. Программа, реализованная на достаточно точно определённом языке программирования, имеет однозначное математическое толкование. Аналогично, требования к программе могут быть выражены на языке математики и логики как точная *спецификация*. Такая формализация открывает возможность строгого *доказательства* соответствия программы и спецификации.

Чтобы лучше ощутить разницу между тестированием и верификацией,

¹Аппаратное обеспечение подвержено схожим проблемам, но мы будем рассматривать исключительно программы.

рассмотрим следующее простое отношение:

$$\sum_{n=0}^N n = \frac{N(N+1)}{2}$$

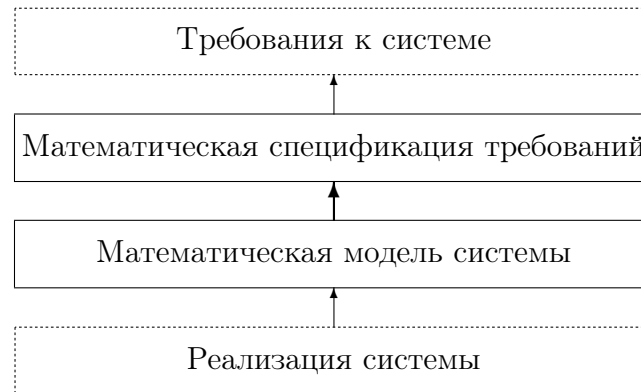
Утверждается, что оно справедливо для произвольных значений N . Мы, очевидно, в состоянии проверить его для любого количества избранных N . Такая проверка может привести нас к интуитивной уверенности в корректности формулы, поскольку выглядит маловероятным, что она когда-либо окажется ложной. Однако, в общем случае попытки сделать выводы по свойствам эмпирических данных могут ввести в заблуждение. В теории чисел известно немало примеров утверждений, которые в итоге опровергались, несмотря на множество положительных частных случаев.

² Более надёжный подход — математическое *доказательство* справедливости приведённого утверждения, что легко достигается путём индукции по переменной N . Аналогично, мы можем надеяться заменить тестирование программы на некотором конечном множестве входных данных формальным доказательством корректности её функционирования в общем случае. В то же время, следует признать, что верификация имеет свои границы применимости.

- Требуется гарантировать, что процесс выполнения программы компьютером в точности соответствует абстрактной математической модели. Возможной причиной расхождений могут стать ошибки в компиляторах или операционных системах, а также физические дефекты аппаратуры. Такие проблемы, безусловно, присущи любым попыткам применения математики в естественных науках. К примеру, предположение точного соответствия между машинной реализацией арифметических операций и их математическими прообразами является естественным упрощением моделей, подобно тому, как при анализе простых динамических систем пренебрегают сопротивлением воздуха. В то же время, оба эти предположения могут оказаться неверными в некоторых случаях.
- Верификация программы опирается на её математическую спецификацию. Возможна ситуация, когда спецификация не соответствует в точности тем требованиям, которые в действительности предъявляются к программе. На самом деле, зачастую оказывается особенно трудно сформулировать математически корректную версию неформальных требований. Имеется достаточное количество свидетельств того, что существенные проблемы компьютерных систем вызваны не ошибками в их реализации, а неверным пониманием ожиданий пользователей. Попытка формализации требований часто бывает полезной сама по себе.

Сложившаяся ситуация может быть представлена диаграммой:

²Например, Литлвуд доказал в 1914 г., что выражение $\pi(n) - li(n)$, где $\pi(n)$ обозначено количество простых чисел, не превышающих n , а $li(n) = \int_0^n du/\ln(u)$, меняет свой знак бесконечное количество раз. Его результат оказался крайней неожиданностью, поскольку в ходе вычисления этого выражения вплоть до $n = 10^{10}$ не было обнаружено ни одной смены его знака.



Мы пытаемся установить связь между верхним и нижним элементами диаграммы, т. е. между требованиями к системе и её реализацией. Чтобы добиться этой цели, нам придётся формализовать и то, и другое. Лишь одна связь на диаграмме, между математическими моделями, обладает математической же точностью, прочие связи остаются неформальными. Всё, что мы можем сделать, это постараться сохранять максимальную простоту и прозрачность правил перехода от неформальных требований (моделей) к формальным и обратно, используя прагматичную модель системы и высокоуровневую достаточно читабельную математическую спецификацию.

Однако, несмотря на эти ограничения, верификация обладает существенными достоинствами. В отличие от тестирования, она устанавливает корректность раз и навсегда, причём, возможно, для целого класса программ одновременно (к примеру, различающихся некоторыми параметрами). Более того, аналитический характер процесса верификации может привести (даже в случае неудачи) к более глубокому пониманию не только программы, но и задачи в целом.

7.1 Функциональные программы как математические объекты

Во введении было отмечено, что (чистые) функциональные программы напрямую соответствуют математическому понятию функции. Из этого факта зачастую делается вывод, что функциональные программы легче поддаются формальному доказательству корректности, чем императивные. Если даже это и справедливо (с чем многие не согласятся), не следует забывать, что разрыв между математической абстракцией и выполнением программы аппаратным обеспечением в случае функционального подхода гораздо больше, чем для типичных императивных языков. В частности, может оказаться, что в глубинах реализации скрываются неприемлемые требования к объёму доступной памяти. Таким образом лёгкость доказательства может достигаться исключением из рассмотрения существенно важных аспектов. Мы не будем обсуждать этот вопрос в данной работе, а сосредоточимся на демонстрации того, что рассуждения о простых функциональных программах зачастую оказываются тривиальными.

В предыдущем разделе был приведён пример, который демонстрирует неточность наивного отождествления функциональных пространств ML с математическими. Однако, если мы не будем углубляться в подробности теории рекурсивных типов и не рассматриваем пространство *произвольных* функций, а ограничимся лишь

некоторыми из них, то в этом случае возможно обращаться с понятиями языка ML, как с математическими.³

Сопоставим функциональным программам математические функции. Располагая уравнениями (обычно рекурсивными), которые определяют функции на языке ML, мы хотим интерпретировать их как аксиоматику соответствующих математических объектов. Например, из определения функции факториала

```
#let rec fact = fun 0 -> 1
                  | n -> n * fact(n - 1);;
```

следует, что $\text{fact}(0) = 1$ и что для произвольных $n \neq 0$ мы имеем $\text{fact}(n) = n * \text{fact}(n - 1)$. Эти утверждения верны, но в дополнение к ним мы должны проявить особое внимание к проблеме завершимости. Для отрицательных значений n программа заикливается, поэтому второе равенство можно полагать истинным лишь в том смысле, что обе его части будут неопределёнными. Анализ всюду определённых функций менее трудоёмок; в противном случае требуется отдельное доказательство завершимости вычислений. В примерах, которые приводятся ниже, это делается параллельно с доказательством корректности, т. е. для каждого аргумента мы должны показать, что вычисления завершаются, причём результат удовлетворяет спецификации.

В общем случае процесс доказательства может потребовать применения самых различных математических приёмов. Однако, очень часто свойства функций (включая завершимость), определённых рекурсивно, можно доказать по индукции в силу двойственности этих двух понятий. Более того, различные формы индукции (арифметическая, структурная, трансфинитная) обычно прямо соответствуют видам рекурсии, применённой в определении. Ниже будут изложены примеры, поясняющие это утверждение.

7.2 Вычисление степени

Напомним нашу простую реализацию вычисления степени:

```
#let rec exp x n =
  if n = 0 then 1
  else x * exp x (n - 1);;
```

Докажем, что `exp` обладает следующим свойством:

Теорема 7.1 *Для любого $n \geq 0$ и произвольного x функция `exp x n` определена и принимает значение x^n .*

Доказательство: Функция `exp` была определена при помощи примитивной пошаговой рекурсии. Как следствие, для доказательства уместно будет воспользоваться пошаговой индукцией. Покажем, что наше утверждение справедливо для $n = 0$, а затем, предположив его справедливость для произвольных $n \geq 0$, докажем его также и для $n + 1$.

³Будем игнорировать возможность арифметического переполнения. CAML поддерживает арифметику произвольной разрядности ценой, в общем случае, неограниченного расхода памяти.

1. При $n = 0$ наша реализация даёт $\text{exp } x \ n = 1$. По определению, для произвольного целого x имеем $x^0 = 1$, таким образом базис индукции выполняется. Заметим, что мы неявно предполагаем справедливость $0^0 = 1$ — хороший пример того, как тщательно нужно составлять спецификации, разрешая неоднозначности (каково значение x^n в общем случае?), при этом учитывая, что некоторые люди могут быть немало удивлены нашим выбором.
2. Предположим, что при $n \geq 0$ мы имеем $\text{exp } x \ n = x^n$. Из $n \geq 0$ следует, что $n + 1 \neq 0$ и, по определению, $\text{exp } x \ (n + 1) = x * \text{exp } x \ ((n + 1) - 1)$. Отсюда

$$\begin{aligned}
 \text{exp } x \ (n + 1) &= x * \text{exp } x \ ((n + 1) - 1) \\
 &= x * \text{exp } x \ n \\
 &= x * x^n \\
 &= x^{n+1}
 \end{aligned}$$

□

7.3 Вычисление НОД

Рассмотрим нашу реализацию вычисления наибольшего общего делителя $\text{gcd}(m, n)$ двух натуральных чисел m и n :

```
#let rec gcd x y =
  if y = 0 then x else gcd y (x mod y);;
```

Будем утверждать, что в действительности эта функция применима к любым целым числам, а не только положительным. Однако, в этом случае нам понадобится расширенное определение понятия НОД. Определим отношение ' $u|v$ ', либо ' u является делителем v ', в которое входят всевозможные пары целых чисел u и v , для которых ' v кратно u ', т. е. найдётся такое целое число d , что $v = du$. Например, $0|0$, $1|11$, $-2|4$, но $0 \nmid 1$, $3 \nmid 5$. Будем говорить, что d является наибольшим общим делителем x и y , если:

- $d|x$ и $d|y$
- Для любых других целых d' , если $d'|x$ и $d'|y$, то $d'|d$.

Отметим, что мы используем в определении $d'|d$, а не $d' \leq d$. Тем самым понятие «наибольший» делитель лишается смысла, но именно так вводится его определение в алгебре. Также заметим, что любая пара чисел (за исключением 0 и 0) имеет два наибольших делителя, поскольку из определения следует, что если число d является НОД x и y , то это же справедливо и для $-d$.

С учётом сказанного выше, наша спецификация примет такой вид: для произвольных целых чисел x и y , $d = \text{gcd } x \ y$ является наибольшим общим делителем x и y . На её примере можно ещё раз убедиться, что разработка спецификации даже такой простой функции зачастую оказывается сложнее,

чем ожидалось. Кроме того, данная спецификация типична ещё и в том, что не определяет однозначно результат вычислений, а лишь задаёт определённые ограничения. Если мы определим функцию `ngcd` как

```
#let rec ngcd x y = -(gcd x y);;
```

то она будет точно так же удовлетворять спецификации, как и `gcd`. Безусловно, мы можем при необходимости сделать спецификацию более строгой. Например, если мы ограничимся положительными значениями x и y , функция будет в полной мере соответствовать понятию наибольшего общего делителя.

Функция `gcd` не определяется исключительно посредством примитивной рекурсии. На шаге алгоритма Евклида `gcd x y` выражается через `gcd y (x mod y)`. Соответственно, вместо пошаговой индукции будет уместно применить трансфинитную. Нам потребуется определить подходящее для этого вида индукции отношение, причём такое, что результаты последующих рекурсивных вызовов предшествуют предыдущим; это гарантирует завершимость алгоритма. В общем случае, для этого могут понадобиться сложные отношения, заданные на аргументах, но часто бывает достаточно придумать меру, которая отображает аргументы на натуральные числа и убывает в ходе рекурсии. В данном случае такой мерой может служить $|y|$.

Теорема 7.2 *Для произвольных целых x и y , вычисление `gcd x y` завершается с результатом, равным НОД x и y .*

Доказательство: *Предположим для некоторого n , что для произвольного значения x и y такого, что $|y| < n$ теорема справедлива. Основываясь на этом предположении, попробуем доказать, что она справедлива также для произвольного x при $|y| = n$. Этого будет достаточно, чтобы считать теорему доказанной, поскольку для любого y найдётся некоторое n такое, что $|y| = n$. Согласно определению функции, доказательство разбивается на два частных случая.*

- *Предположим, что $y = 0$. В этом случае по определению функции `gcd x y = x`. Очевидно, $x|x$ и $x|0$, т. е. является общим делителем. Предположим, что найдётся ещё один общий делитель d , для которого справедливо $d|x$ и $d|0$. Отсюда немедленно следует $d|x$, из чего, в свою очередь, вытекает то, что x является НОД.*
- *Рассмотрим случай $y \neq 0$. Мы хотим применить индуктивное предположение к `gcd y (x mod y)`. Введём сокращённое обозначение $r = x \bmod y$. Основным свойством функции `mod` является то, что при $y \neq 0$ найдётся такое целое q , что $x = qy + r$ и $|r| < |y|$. Поскольку $|r| < |y|$, из индуктивного предположения следует, что $d = \text{gcd } y (x \bmod y)$ является НОД y и r . Остается показать, что он является НОД также и для x и y . Очевидно, это также справедливо, поскольку при $d|y$ и $d|r$ мы имеем $d|x$, так как $x = qy + r$. Предположим, что $d'|x$ и $d'|y$. Аналогично сказанному выше, обнаруживаем, что $d'|r$. Таким образом, d' является общим делителем y и r , а по индуктивному предположению $d'|d$, что и требовалось.*

□

Отметим, что в доказательстве было использовано основное свойство операции вычисления остатка. Её реализация (функция `mod`) в конкретных САМЛ-системах требует тщательной проверки соответствия теоретическому определению. Хорошо известно, что различные языки программирования (и даже реализации одного и того же языка) зачастую различаются в трактовке операции вычисления остатка в случае отрицательных аргументов. Если отсутствует уверенность в надёжности неявных предположений, их можно добавить в теорему в явной форме.

7.4 Конкатенация списков

Рассмотрим пример верификации функции, оперирующей списками. Функция `append` предназначена, как это следует из её названия, для конкатенации (сцепления) двух списков. Например, результатом применения этой операции к спискам `[3; 2; 5]` и `[6; 3; 1]` будет `[3; 2; 5; 6; 3; 1]`.

```
#let rec append l1 l2 =
  match l1 with
  [] -> l2
  | (h::t) -> h::(append t l2);;
```

Данная функция вводится при помощи примитивной рекурсии в соответствии с определением списочного типа. Она определяется для случая пустого списка и, затем, для списка вида $h :: t$, причём в последнем случае используется значение этой же функции для аргумента t . Следовательно, доказательства теорем о её свойствах удобно строить на соответствующем принципе структурной индукции для списков: если некоторое утверждение справедливо для пустого списка и, если из предположения, что оно выполняется для t следует, что данное утверждение справедливо и для $h :: t$, то мы можем заключить, что оно справедливо для любого списка. Однако, такой подход не обязателен — при желании можно воспользоваться математической индукцией по длине списка. Докажем с учётом сказанного выше, что операция конкатенации ассоциативна.

Теорема 7.3 *Для трёх произвольных списков l_1 , l_2 и l_3 справедливо:*

$$\text{append } l_1 (\text{append } l_2 l_3) = \text{append } (\text{append } l_1 l_2) l_3$$

Доказательство: *Применяя структурную индукцию к l_1 , докажем, что требуемое свойство выполняется для произвольных l_2 и l_3 .*

- Если $l_1 = []$, то:

$$\begin{aligned} \text{append } l_1 (\text{append } l_2 l_3) &= \text{append } [] (\text{append } l_2 l_3) \\ &= \text{append } l_2 l_3 \\ &= \text{append } (\text{append } [] l_2) l_3 \\ &= \text{append } (\text{append } l_1 l_2) l_3 \end{aligned}$$

- Рассмотрим случай $l_1 = h :: t$. Предположим, что для l_2 и l_3 мы имеем

$$\text{append } t (\text{append } l_2 l_3) = \text{append } (\text{append } t l_2) l_3$$

Отсюда следует

$$\begin{aligned} \text{append } l_1 (\text{append } l_2 l_3) &= \text{append } (h :: t) (\text{append } l_2 l_3) \\ &= h :: (\text{append } t (\text{append } l_2 l_3)) \\ &= h :: (\text{append } (\text{append } t l_2) l_3) \\ &= \text{append } (h :: (\text{append } t l_2)) l_3 \\ &= \text{append } (\text{append } (h :: t) l_2) l_3 \\ &= \text{append } (\text{append } l_1 l_2) l_3 \end{aligned}$$

□

7.5 Обращение списков

Определить функцию обращения списков несложно:

```
#let rec rev =
  fun [] -> []
    | (h::t) -> append (rev t) [h];;
rev : 'a list -> 'a list = <fun>
#rev [1;2;3];;
- : int list = [3; 2; 1]
```

Докажем, что функция `rev` является инволюцией, т. е. что

$$\text{rev}(\text{rev } l) = l$$

Однако, если мы попытаемся напрямую применить структурную индукцию, оказывается, что нам понадобится сперва доказать пару дополнительных лемм.

Лемма 7.4 Для произвольного списка l справедливо $\text{append } l [] = l$.

Доказательство: Воспользуемся структурной индукцией по l .

- Для $l = []$ имеем:

$$\begin{aligned} \text{append } l [] &= \text{append } [] [] \\ &= [] \\ &= l \end{aligned}$$

- Пусть $l = h :: t$. Предполагая, что $\text{append } t [] = t$, получим

$$\begin{aligned} \text{append } l [] &= \text{append } (h :: t) [] \\ &= h :: (\text{append } t []) \\ &= h :: t \\ &= l \end{aligned}$$

□

Лемма 7.5 Для произвольных списков l_1 и l_2 справедливо

$$\text{rev}(\text{append } l_1 \ l_2) = \text{append } (\text{rev } l_2) \ (\text{rev } l_1)$$

Доказательство: Для доказательства этой леммы также воспользуемся структурной индукцией по l_1 .

- При $l_1 = []$ получаем

$$\begin{aligned} \text{rev}(\text{append } l_1 \ l_2) &= \text{rev}(\text{append } [] \ l_2) \\ &= \text{rev } l_2 \\ &= \text{append } (\text{rev } l_2) [] \\ &= \text{append } (\text{rev } l_2) (\text{rev } []) \end{aligned}$$

- Если $l_1 = h :: t$ и мы знаем, что

$$\text{rev}(\text{append } t \ l_2) = \text{append } (\text{rev } l_2) \ (\text{rev } t)$$

то из этого следует

$$\begin{aligned} \text{rev}(\text{append } l_1 \ l_2) &= \text{rev}(\text{append } (h :: t) \ l_2) \\ &= \text{rev}(h :: (\text{append } t \ l_2)) \\ &= \text{append } (\text{rev}(\text{append } t \ l_2)) [h] \\ &= \text{append } (\text{append } (\text{rev } l_2) (\text{rev } t)) [h] \\ &= \text{append } (\text{rev } l_2) (\text{append } (\text{rev } t) [h]) \\ &= \text{append } (\text{rev } l_2) (\text{rev } (h :: t)) \\ &= \text{append } (\text{rev } l_2) (\text{rev } l_1) \end{aligned}$$

□

Теорема 7.6 Для произвольного списка l справедливо $\text{rev}(\text{rev } l) = l$.

Доказательство: Применим структурную индукцию по l .

- При $l = []$ имеем:

$$\begin{aligned} \text{rev}(\text{rev } l) &= \text{rev}(\text{rev } []) \\ &= \text{rev } [] \\ &= [] \\ &= l \end{aligned}$$

- Пусть $l = h :: t$ и справедливо, что

$$\text{rev}(\text{rev } t) = t$$

в этом случае мы получим:

$$\begin{aligned}
 \text{rev}(\text{rev } l) &= \text{rev}(\text{rev } (h :: t)) \\
 &= \text{rev}(\text{append } (\text{rev } t) [h]) \\
 &= \text{append } (\text{rev } [h]) (\text{rev}(\text{rev } t)) \\
 &= \text{append } (\text{rev } [h]) t \\
 &= \text{append } (\text{rev } (h :: [])) t \\
 &= \text{append } (\text{append } (\text{rev } []) [h]) t \\
 &= \text{append } (\text{append } [] [h]) t \\
 &= \text{append } [h] t \\
 &= \text{append } (h :: []) t \\
 &= h :: (\text{append } [] t) \\
 &= h :: t \\
 &= l
 \end{aligned}$$

□

Большое количество теорем о свойствах операций над списками может быть доказано в аналогичном стиле. Доказательство в целом базируется на структурной индукции, применяемой к списку. В некоторых случаях может оказаться удобным выделить части рассуждений в отдельные леммы, которые также доказываются индуктивно, кроме того, задача может потребовать обобщения, прежде чем индукция окажется применимой. Некоторые примеры таких задач приведены в упражнениях.

Дополнительная литература

В работе [Neumann \(1995\)](#) приведён обзор рисков, связанных как с распространением компьютеров в обществе в целом, так и с последствиями программных ошибок в частности. Очень интересная дискуссия с примерами на эту же тему даётся в [Peterson \(1996\)](#). Применимость верификации являлась в своё время предметом споров; некоторые аргументы против этого подхода изложены в [DeMillo, Lipton, and Perlis \(1979\)](#). Также заслуживает внимания обоснованное обсуждение [Barwise \(1989\)](#). На данный момент количество публикаций по верификации велико, но в большинстве своём они рассматривают императивные программы. Некоторые учебники функционального программирования для начинающих, такие как [Paulson \(1991\)](#) и [Reade \(1989\)](#), включают базовые примеры наподобие рассмотренных в данной книге. Представляет интерес работа [Boyer and Moore \(1979\)](#), авторы которой провели верификацию свойств определений чистых функций на языке LISP при помощи своей системы автоматизированного доказательства теорем. Одним из самых масштабных опытов по верификации, наподобие рассмотренных здесь, служит применение данной методики к программе упрощения логических выражений, которая используется в синтезе СБИС [Aagaard and Leeser \(1994\)](#).

Упражнения

1. Докажите корректность более эффективного алгоритма возведения в степень:

```
#let square x = x * x;;
#let rec exp x n =
  if n = 0 then 1
  else if n mod 2 = 0 then square(exp x (n / 2))
  else x * square(exp x (n / 2));;
```

2. Пусть функция `length` определена как

```
#let rec length =
  fun [] -> 0
    | (h::t) -> 1 + length t;;
```

Докажите, что $\text{length}(\text{rev } l) = \text{length } l$ и что $\text{length}(\text{append } l_1 \ l_2) = \text{length } l_1 + \text{length } l_2$.

3. Определим функцию `map`, которая применяет заданную функцию к каждому элементу списка, следующим образом:

```
#let rec map f =
  fun [] -> []
    | (h::t) -> (f h)::(map f t);;
```

Докажите, что если $l \neq []$ то $\text{hd}(\text{map } f \ l) = f(\text{hd } l)$ и $\text{map } f \ (\text{rev } l) = \text{rev } (\text{map } f \ l)$. Далее, используя следующее определение композиции функций,

```
#let o f g = fun x -> f(g x);;
#infix "o";;
```

докажите, что $\text{map } f \ (\text{map } g \ l) = \text{map } (f \circ g) \ l$.

4. Функция «91» Мак-Карти может быть определена так:

```
#let rec f x = if x > 100 then x - 10
               else f(f(x + 11));;
```

Докажите, что для $n \leq 101$ справедливо $f(n) = 91$. Особое внимание следует уделить доказательству завершимости. (Указание: возможно использовать меру $101 - x$.)

5. Задача о голландском флаге состоит в сортировке списка «цветов» (красный, белый, синий) так, чтобы цвета расположились в данном порядке. Требуется решить задачу, используя исключительно перестановку соседних элементов. Пример решения на языке ML приводится ниже. Функция `dnf` возвращает значение `true` тогда и только тогда, когда порядок следования элементов изменился; её вычисление повторяется, пока на некотором этапе не окажется, что изменений не обнаружено.

```

type colour = Red | White | Blue;;

let rec dnf =
  fun [] -> [], false
    | (White::Red::rest) -> Red::White::rest, true
    | (Blue::Red::rest) -> Red::Blue::rest, true
    | (Blue::White::rest) -> White::Blue::rest, true
    | (x::rest) -> let fl, ch = dnf rest in x::fl, ch;;

let rec flag l =
  let l', changed = dnf l in
  if changed then flag l' else l';;

```

Например,

```

#flag [White; Red; Blue; Blue; Red; White; White; Blue; Red];;
- : colour list =
  [Red; Red; Red; White; White; White; Blue; Blue; Blue]

```

Докажите, что функция `flag` всегда завершается, и её результатом будет корректно упорядоченный список.

6. (*) Определите следующие функции:

```
#let rec sorted =  
  fun [] -> true  
  | [h] -> true  
  | (h1::h2::t) -> h1 <= h2 & sorted(h2::t);;  
  
#let rec filter p =  
  fun [] -> []  
  | (h::t) -> let t' = filter p t in  
               if p h then h::t' else t';;  
  
#let sameprop p l1 l2 =  
  length(filter p l1) = length(filter p l2);;  
  
#let rec permutes l1 l2 =  
  fun [] -> true  
  | (h::t) -> sameprop (fun x -> x = h) l1 l2 &  
               permutes l1 l2 t;;  
  
#let permuted l1 l2 = permutes l1 l2 l1;;
```

Определите назначение этих функций. Реализуйте функцию сортировки `sort` и докажите, что для произвольных списков l справедливо, что $\text{sorted}(\text{sort } l) = \text{true}$ and $\text{permuted } l (\text{sort } l) = \text{true}$.

Глава 8

Эффективный ML

В этой главе мы обсудим некоторые техники и уловки, используя которые, ML программисты делают свои программы более элегантными и эффективными. Затем мы рассмотрим дополнительные *императивные* возможности, которые могут быть задействованы, когда чистый функциональный подход представляется неподходящим.

8.1 Полезные комбинаторы

Гибкость функций высшего порядка означает, что можно написать небольшие, но весьма полезные функции, а затем многократно использовать для множества сходных задач. Такие функции часто называют *комбинаторами*, и не только потому, что они являются лямбда-термами без свободных переменных. Эти функции зачастую оказываются настолько гибкими, что путём их совместной комбинации можно реализовать практически всё что угодно.

Например, очень полезный комбинатор для операций над списками, часто называемый `itlist` или `fold`, выполняющий следующие действия:

$$\text{itlist } f [x_1; x_2; \dots; x_n] b = f x_1 (f x_2 (f x_3 (\dots (f x_n b))))$$

Простое определение на ML:

```
#let rec itlist f =  
  fun [] b -> b  
    | (h::t) b -> f h (itlist f t b);;  
itlist : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Довольно часто рекурсивные функции на списках заключаются просто в повторном применении некоторой функции к каждому элементу списка. Используя `itlist` с подходящим аргументом, такие функции можно очень легко реализовать без явного использования рекурсии. Типичный пример – нахождение суммы всех чисел в списке.

```
#let sum l = itlist (fun x sum -> x + sum) l 0;;
sum : int list -> int = <fun>
#sum [1;2;3;4;5];;
- : int = 15
#sum [];;
- : int = 0
#sum [1;1;1;1];;
- : int = 4
```

Читатели, заинтересованные в краткости, могут предпочесть следующий код:

```
#let sum l = itlist (prefix +) l 0;;
```

Легко изменить эту функцию для вычисления произведения, вместо суммы.

```
#let prod l = itlist (prefix *) l 1;;
```

Многие полезные функции над списками могут быть реализованы подобным образом. Например, вот функция, которая отбирает только те элементы списка, которые удовлетворяют предикату:

```
#let forall p l = itlist (fun h a -> p(h) & a) l true;;
forall : ('a -> bool) -> 'a list -> bool = <fun>
#let exists p l = itlist (fun h a -> p(h) or a) l false;;
exists : ('a -> bool) -> 'a list -> bool = <fun>
#forall (fun x -> x < 3) [1;2];;
- : bool = true
#forall (fun x -> x < 3) [1;2;3];;
- : bool = false
```

Эти функции проверяют наличие в списке элементов, удовлетворяющих предикату:

```
#let forall p l = itlist (fun h a -> p(h) & a) l true;;
forall : ('a -> bool) -> 'a list -> bool = <fun>
#let exists p l = itlist (fun h a -> p(h) or a) l false;;
exists : ('a -> bool) -> 'a list -> bool = <fun>
#forall (fun x -> x < 3) [1;2];;
- : bool = true
#forall (fun x -> x < 3) [1;2;3];;
- : bool = false
```

А вот альтернативные версии `length`, `append` и `map`:

```
#let length l = itlist (fun x s -> s + 1) l 0;;
length : 'a list -> int = <fun>
#let append l m = itlist (fun h t -> h::t) l m;;
append : 'a list -> 'a list -> 'a list = <fun>
#let map f l = itlist (fun x s -> (f x)::s) l [];;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Некоторые из этих функций сами по себе являются полезными комбинаторами. Например, если мы хотим рассматривать списки как множества, т.е. исключаем кратные элементы, то многие стандартные операции на множествах очень просто выражаются в терминах только что рассмотренных комбинаторов:

```
#let mem x l = exists (fun y -> y = x) l;;
mem : 'a -> 'a list -> bool = <fun>
#let insert x l =
  if mem x l then l else x::l;;
insert : 'a -> 'a list -> 'a list = <fun>
#let union l1 l2 = itlist insert l1 l2;;
union : 'a list -> 'a list -> 'a list = <fun>
#let setify l = union l [];
setify : 'a list -> 'a list = <fun>
#let Union l = itlist union l [];
Union : 'a list list -> 'a list = <fun>
#let intersect l1 l2 = filter (fun x -> mem x l2) l1;;
intersect : 'a list -> 'a list -> 'a list = <fun>
#let subtract l1 l2 = filter (fun x -> not mem x l2) l1;;
subtract : 'a list -> 'a list -> 'a list = <fun>
#let subset l1 l2 = forall (fun t -> mem t l2) l1;;
subset : 'a list -> 'a list -> bool = <fun>
```

Функция `setify` предназначена для перевода списка в множество, посредством исключения кратных элементов.

8.2 Создание эффективного кода

Здесь мы собрали несколько общих приёмов, с помощью которых можно сделать ML программы существенно более эффективными. Для того, чтобы оправдать некоторые из них, необходимо обрисовать в общих словах, как определённые конструкции выполняются на «железе».

8.2.1 Хвостовая рекурсия и аккумуляторы

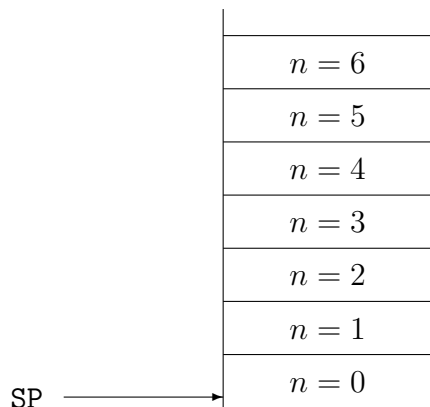
Основной механизм управления в функциональных программах – рекурсия. Если мы заинтересованы в эффективных программах, нам следует подумать немного о том, как рекурсия реализуется на стандартном оборудовании. В сущности, нет или, по крайней мере, мало отличий между реализацией ML и многими другими языками с динамическими переменными, такими как C.

В языках, не допускающих рекурсивных вызовов можно без опаски сохранять все локальные переменные (включая и значения аргументов функции) в фиксированной области памяти, так, например, сделано в FORTRAN. Тем не менее, в общем случае, когда функция может вызываться рекурсивно, это не допустимо: вызов функции `f` с одним набором аргументов содержит вызов `f`, возможно, с другими аргументами. Это означает, что старые значения аргументов будут переписаны, даже если они могут потребоваться в дальнейшем, после вызова вложенной `f`. Поясним сказанное на примере. Рассмотрим ещё раз функцию факториала:

```
#let rec fact n = if n = 0 then 1
                  else n * fact(n - 1);;
```

Вызов `fact 6` приводит к последующему вызову `fact 5` (и далее), но когда вызов `fact 5` закончен, и получен результат, нам всё ещё требуется исходное значение `n`, а именно 6, для произведения, дающего окончательный ответ. Обычно в реализациях

поступают следующим образом: для каждого вызова функции выделяется новый *стековый фрейм*. И каждый новый вызов функции передвигает указатель стека далее вниз¹ стека, выделяя пространство под новые переменные. В стековый фрейм и копируются значения аргументов функции, её локальных переменных, иначе – *состояние выполнения*. После того, как вызванная функция завершает выполнение, указатель на стек перемещается обратно, таким образом, ненужные внутренние переменные автоматически отбрасываются. Схема может прояснить сказанное:



Здесь изображён воображаемый снимок стека во время последнего рекурсивного вызова, т.е. вычисления **fact 0**. Все локальные переменные от предыдущих вызовов собраны в стеке, с каждым экземпляром функции, имеющим свой фрейм. По завершению вызовов указатель стека **SP** перемещается обратно вверх.

Следовательно, наша реализация **fact** требует n фреймов стека для вычисления на аргументе n . Напротив, рассмотрим следующую реализацию функции, вычисляющей факториал:

```
#let rec tfact x n =
  if n = 0 then x
  else tfact (x * n) (n - 1);;
tfact : int -> int -> int = <fun>
#let fact n = tfact 1 n;;
fact : int -> int = <fun>
#fact 6;;
- : int = 720
```

Хотя **tfact** также рекурсивна, здесь рекурсивный вызов – итоговое выражение, после него никаких вычислений не происходит; оно не является подвыражением, частью другого выражения. Подобные вызовы называются *хвостовыми вызовами* (потому что самые последние в теле функции), а функция, в которой все рекурсивные вызовы суть хвостовые, называется функцией с *хвостовой рекурсией*.

Что знаменательно в хвостовых вызовах? При рекурсивном вызове **tfact** нет необходимости сохранять предыдущие значения локальных переменных. Можно использовать одну и ту же фиксированную область памяти для хранения переменных. Будет ли это происходить на самом деле, разумеется, зависит от того, насколько компилятор способен обнаружить наличие хвостовой рекурсии в коде. Большинство известных компиляторов, включая CAML, хвостовую

¹Такая уж сложилась традиция – стек "растёт" вниз.

рекурсию обнаруживают. Следовательно, оформление функции таким образом, что рекурсивный вызов является хвостовым, может существенно сократить использование памяти. Для таких функций как факториал, едва ли возможен вызов на столь больших значениях аргументов n , что переполнится стек. Однако, наивная реализация многих функций над списками, может привести к подобному результату, когда списки длинны.

Дополнительный аргумент x функции `tfact` называется *аккумулятором*, потому что он накапливает промежуточные результаты по ходу рекурсивных вызовов и, в конце концов, возвращается как значение функции. Оформление кода таким способом – самый простой способ сделать рекурсивную функцию хвостовой рекурсией.

Мы отметили, что в функциях с хвостовой рекурсией для аргументов может использоваться фиксированная область памяти. С этой точки зрения можно рассматривать рекурсию как завуалированную реализацию императивного цикла. Вот очевидная параллель с реализацией факториала на C:

```
int fact(int n)
{ int x = 1;
  while (n > 0)
  { x = x * n;
    n = n - 1;
  }
  return x;
}
```

Инициализация $x = 1$ соответствует присваиванию 1 переменной x . Основным циклом соответствует рекурсивным вызовам, с тем лишь отличием, что для функции с хвостовой рекурсией мы явно передаём состояние² через аргументы. Вместо присваивания и введения циклов, мы делаем рекурсивный вызов с обновлёнными переменными. Используя похожие приёмы и выражая состояние явно, можно легко писать, в сущности, императивный код в якобы функциональном стиле. Мы ведь знаем, после оптимизаций стандартными компиляторами машинный код будет один и тот же в обоих случаях.

8.2.2 Минимизация операций `cons`

Мы рассмотрели, как используется пространство стековой памяти. Но различные конструкции в функциональных программах могут использовать память и другого типа, память, обычно выделяемую в области, называемой *кучей*. Тогда как стек последовательным образом растёт и сокращается, управляемый последовательностью вызовов функций, эта память не может быть освобождена таким же простым способом. Вместо этого исполняющей системе, время от времени, требуется проверять, какая часть выделенной памяти больше не используется, и освобождать её для дальнейших вычислений, этот процесс известен как *сборка мусора*. Особенно важный пример здесь – это память, используемая конструкторами рекурсивных типов, например, при вызове `::`. Так, когда следующий фрагмент выполняется:

²Под «состоянием» здесь понимается набор переменных, которые меняются по ходу цикла.

```
let l = 1::[] in tl l;;
```

новый блок памяти, именуемый «cons-ячейка», выделяется для хранения результата работы конструктора `::`. Обычно это три слова памяти, одно является идентификатором конструктора, а два других являются указателями на голову и хвост списка. В общем случае, решение по поводу того, когда память может быть освобождена, представляет собой сложную задачу. Для нашего примера, мы сразу возвращаем хвост списка, так что, очевидно, cons-ячейка может быть освобождена немедленно. Но, в общем случае, из текста программы столь очевидные выводы сделать затруднительно, поскольку `l` может быть, например, передан различным функциям, которые, могут просматривать элементы списка. Так что необходимо динамически анализировать использование памяти и выполнять сборку мусора. В противном случае мы рискуем исчерпать память.

Разработчики функциональных языков усердно работают над алгоритмами более эффективной сборки мусора. Некоторые заявляют, что автоматическое выделение памяти и сборка мусора работают быстрее, чем в обычном использовании явного выделения памяти в языках подобных *C* (`malloc` и пр.). Хотя мы не пойдём так далеко, конечно, хорошо, что память всегда выделяется автоматически. Это позволяет избежать многих тех моментов в программировании, которые печально известны своей утомительностью и подверженностью к ошибкам.

Многие конструкции, излюбленные функциональными программистами, используют память, а она управляется сборщиком мусора. Хотя чрезмерное внимание к этому может повредить функциональному стилю программ, есть некоторые простые соображения, которые стоит учитывать во избежание неуместного расходования памяти. Очень простое правило заключается в том, следует избегать использования `append` если возможно. Как видим, при развёртывании рекурсивных вызовов согласно определению

```
#let rec append l1 l2 =
  match l1 with
  [] -> l2
  | (h::t) -> h::(append t l2);;
```

порождает n cons-ячеек, где n длина первого списка. Есть много способов заменить `append`, например, с помощью добавления к функциям дополнительных аргументов – аккумуляторов. Замечательный пример – функция обращения списка, которую мы раньше записывали как:

```
#let rec rev =
  fun [] -> []
  | (h::t) -> append (rev t) [h];;
```

Что порождает порядка $n^2/2$ cons-ячеек, где n это длина списка. Следующая альтернатива, используя аккумулятор, генерирует только n из них:

```
#let rev =
  let rec reverse acc =
    fun [] -> acc
    | (h::t) -> reverse (h::acc) t in
  reverse [];;
```

Более того, ядро рекурсии `reverse` – хвостовая рекурсия, так что мы также сохраняем стековую память и, таким образом, дважды в выигрыше.

Как пример другой типичной ситуации, когда посредством разумного использования аккумуляторов мы можем избежать вызовов `append`, рассмотрим задачу по нахождению списка терминальных элементов бинарного дерева. Если мы зададим тип бинарных деревьев следующим образом:

```
#type btree = Leaf of string
           | Branch of btree * btree;;
```

то решением задачи будет:

```
#let rec fringe =
  fun (Leaf s) -> [s]
    | (Branch(l,r)) -> append (fringe l) (fringe r);;
```

Однако следующая улучшенная версия выполняется с меньшими затратами:

```
#let fringe =
  let rec fr t acc =
    match t with
    (Leaf s) -> s::acc
    | (Branch(l,r)) -> fr l (fr r acc) in
  fun t -> fr t [];;
```

Заметьте, мы переписали второй аргумент – аккумулятор, так что теперь рекурсивный вызов более понятен при чтении слева - направо. Вот простой пример, как может использоваться каждая из версий `fringe`:

```
#fringe (Branch(Branch(Leaf "a",Leaf "b"),
                 Branch(Leaf "c",Leaf "d")));;
- : string list = ["a"; "b"; "c"; "d"]
```

Первая версия создаёт 6 cons-ячеек, вторая – только 4. На больших деревьях разность будет более впечатляющей. Ещё неуместное использование конструкторов типа происходит при сопоставлении с образцом. Например, рассмотрим фрагмент кода:

```
fun [] -> []
  | (h::t) -> if h < 0 then t else h::t;;
```

Ветка, соответствующая ‘else’ создаёт cons-ячейку, несмотря на то, что эта же конструкция уже передавалась как аргумент функции. То есть, аргумент берётся и повторно реконструируется. Чтобы подобных издержек не было, следует кодировать функцию следующим образом:

```
fun l ->
  match l with
  [] -> []
  | (h::t) -> if h < 0 then t else l;;
```

Тем не менее, ML предлагает более гибкую альтернативу: использование ключевого слова **as**. С его помощью мы можем именовать определённые части образцов, так что впоследствии возможно их использование без реконструкции. Например:

```
fun [] -> []  
  | (h::t as l) -> if h < 0 then t else l;;
```

8.2.3 Принудительное вычисление

Мы отмечали, что можно использовать лямбда-абстракцию для приостановки вычислений, так как ML не вычисляет выражения под лямбдами. Некоторые интересные примеры мы рассмотрим позже. С другой стороны, кому-то может потребоваться принудительное вычисление выражений. Например:

```
#let rec tfact x n =  
  if n = 0 then x  
  else tfact (x * n) (n - 1);;  
#let fact n = tfact 1 n;;
```

Поскольку мы в действительности никогда непосредственно **tfact** не используем, представляется нецелесообразным закреплять имя за этой функцией. Взамен мы можем сделать её локальной:

```
#let fact1 n =  
  let rec tfact x n =  
    if n = 0 then x  
    else tfact (x * n) (n - 1) in  
  tfact 1 n;;
```

Здесь, однако, недостаток: локальное рекурсивное определение вычисляется только после того, как **fact1** получает свой аргумент, поскольку до этого оно находится под лямбдой. Более того, оно вычисляется каждый раз, при вызове **fact**. Мы можем поправить это так:

```
#let fact2 =  
  let rec tfact x n =  
    if n = 0 then x  
    else tfact (x * n) (n - 1) in  
  tfact 1;;
```

Теперь локальное связывание вычисляется лишь один раз, во время объявления **fact2**. В соответствии с нашими испытаниями, вторая версия **fact** примерно на 20% быстрее, при заданном аргументе 6. В тех случаях, когда при локальном связывании требуется больше вычислений, отличие может быть более впечатляющим. На самом деле, ведутся сложные исследования в области «частичных вычислений» посвящённые этим, а также более изощрённым оптимизациям. В известном смысле, это обобщение стандартных оптимизаций, таких как «сворачивание констант» для компилятора обычных языков. Однако в рабочих системах ML ответственность за

принудительные вычисления обычно возлагается на пользователя, как в примере выше.

Мимоходом заметим, что для функций собранных из совокупности комбинаторов, без навешенных лямбд, больше возможностей во время объявления вычислить настолько большую часть выражения, насколько возможно. То есть, вот простейший пример: $f \circ g$ выполнит все вычисления возможные в f и g , тогда как $\lambda x. f(g\ x)$ не будет вычислено вообще, до тех пор, пока не получит аргумента. С другой стороны, когда мы действительно *хотим* отложить вычисление, нам, действительно, нужна лямбда-абстракция, так что исключительно комбинаторная версия невозможна.

8.3 Императивные возможности

ML обладает довольно широкими возможностями для императивного программирования. Мы не будем тратить время на императивный стиль программирования, поскольку не это цель нашего курса, и мы предполагаем, что у читателя уже есть достаточно опыта. Следовательно, мы бегло рассмотрим эти вопросы с несколькими иллюстрирующими примерами. Тем не менее, некоторые императивные возможности будут использоваться далее в примерах, и некоторое знание того, что можно использовать будет хорошим подспорьем при написании настоящего ML кода.

8.3.1 Исключения

Мы знаем, что иногда в вычислениях происходят исключительные ситуации, и вычисление не завершается правильным образом. Например, вследствие неудачных сопоставлений с образцом. Есть и другие причины для критических ошибок, например попытка деления на ноль.

```
#1 / 0;;  
Uncaught exception: Division_by_zero
```

Во всех этих случаях компилятор сообщает о «необработанном исключении». Исключение – это признак ошибки, индикация того, что вычисления пошли неправильным путём, при этом, часто бывает, ошибки можно обработать локально. Для *исключений* есть тип `exn`, который фактически является рекурсивным типом, хотя обычно он рекурсивный только номинально. В отличие от обычных типов, для типа `exn` допустимо вводить конструкторы в любом месте программы, используя объявления исключений, например:

```
#exception Died;;  
Exception Died defined.  
#exception Failed of string;;  
Exception Failed defined.
```

В то время как обычно исключения генерирует определённая совокупность операций, этого же можно добиться прямым использованием конструктора `raise`, например:

```
#raise (Failed "I don't know why");;
Uncaught exception: Failed "I don't know why"
```

Мы можем создать своё собственное исключение на случай попытки взять первый элемент в пустом списке:

```
#exception Head_of_empty;;
Exception Head_of_empty defined.
#let hd = fun [] -> raise Head_of_empty
          | (h::t) -> h;;
hd : 'a list -> 'a = <fun>
#hd [];;
Uncaught exception: Head_of_empty
```

Обычно исключения последовательно передаются «наверх»³, но они, также, могут быть «перехвачены» и обработаны в окружении функции, с помощью конструкции `try ...with`, сопровождаемой вариантами исключений:

```
#let headstring sl =
  try hd sl
  with Head_of_empty -> ""
    | Failed s -> "Failure because " ^ s;;
headstring : string list -> string = <fun>
#headstring ["hi"; "there"];;
- : string = "hi"
#headstring [];;
- : string = ""
```

Являются ли исключения императивной возможностью это, в действительности, вопрос убеждений. С одной стороны, можно считать, что функции возвращают элементы составного типа – собственно из возвращаемого типа, декларируемого явно, и типа исключений. С другой стороны, исключения предлагают извращённое, не локальное управление потоком исполнения, наподобие `goto`.⁴ Какой бы смысл в них не вкладывался, исключения часто могут быть весьма полезны.

8.3.2 Ссылки и массивы

В ML есть настоящие переменные, к которым можно присваивать, и выражения могут в качестве побочного эффекта изменять значения этих переменных. Доступ к переменным осуществляется с помощью *ссылок* (указателей, говоря языком C), и ссылки в свою очередь рассматриваются в ML как обычные значения. Действительно, этот подход довольно похож на использование указателей в C. Например, в C, когда требуется использовать «изменяемые параметры» – параметры, для которых допустимо изменение значений после вычисления функции, используется передача параметров по указателю. Подобная техника часто используется, когда функция должна возвращать сложные составные данные.

В ML, запись `ref x` означает объявление и инициализацию ячейки памяти значением `x`. Инициализация обязательна. Это выражение выдаёт ссылку

³По стеку вызовов функций

⁴Возможно, более подходящим примером будут `setjmp` и `longjmp` из C.

(указатель) на ячейку в памяти. Последовательный доступ к содержимому памяти требует явного разыменования указателя с помощью оператора `!`, сходного с унарным `*` в C. Вот как это делается:

```
#let x = ref 1;;
x : int ref = ref 1
#!x;;
- : int = 1
#x := 2;;
- : unit = ()
#!x;;
- : int = 2
#x := !x + !x;;
- : unit = ()
#x;;
- : int ref = ref 4
#!x;;
- : int = 4
```

Заметьте, что во многих отношениях `ref` ведёт себя подобно конструктору типа, а значит, может использоваться в сопоставлении с образцом. Следовательно, оператор разыменования `!` можно было бы определить как:

```
#let contents_of (ref x) = x;;
contents_of : 'a ref -> 'a = <fun>
#contents_of x;;
- : int = 4
```

Будучи изменяемыми, ссылки иногда полезны для создания различных структур данных. Можно легко создавать структуры типа граф, с узлами, содержащими указатель на подграф.

Кроме отдельных ячеек, также в ML можно использовать массивы. В CAML они называются **векторами**. Массив элементов типа α имеет тип $\alpha\ vect$. Новый вектор размера n , где каждый элемент проинициализирован значением x (опять же, инициализация обязательна) создаётся с помощью следующего вызова:

```
#make_vect n x;;
```

Можно прочесть элемент m вектора v с помощью:

```
#vect_item v m;;
```

а записать значение y в m -й элемент v :

```
#vect_assign v m y;;
```

Эти операции соответствуют выражениям $v[m]$ и $v[m] = y$ в C. Элементы массива нумеруются с нуля. Например:

```
#let v = make_vect 5 0;;
v : int vect = [|0; 0; 0; 0; 0|]
#vect_item v 1;;
- : int = 0
#vect_assign v 1 10;;
- : unit = ()
#v;;
- : int vect = [|0; 10; 0; 0; 0|]
#vect_item v 1;;
- : int = 10
```

Все операции чтения и записи элементов сопровождаются контролем границ, например:

```
#vect_item v 5;;
Uncaught exception: Invalid_argument "vect_item"
```

8.3.3 Последовательность вычислений

В ML нет необходимости указывать последовательность вычислений, поскольку обычные правила вычисления предполагают порядок. Например:

```
#let _ = x := !x + 1 in
let _ = x := !x + 1 in
let _ = x := !x + 1 in
let _ = x := !x + 1 in
();;
```

выражения вычисляются в ожидаемом порядке. Здесь мы используем специальный образец `_`, который не привязывает значение к конкретному имени. Тем не менее, более заманчиво использовать определённые обозначения для установления последовательности вычислений. В ML это возможно с помощью точки с запятой:

```
#x := !x + 1;
x := !x + 1;
x := !x + 1;
x := !x + 1;;
```

8.3.4 Работа с системой типов

Хотя полиморфизм очень хорош для чисто функциональной части ML, взаимодействие с некоторыми императивными возможностями построено неудачно. Например, рассмотрим следующее:

```
#let l = ref [];;
```

Может показаться `l` имеет полиморфный тип $\alpha \text{ list ref}$. В соответствии со стандартными правилами вывода по полиморфизму, мы могли бы использовать `l` с двумя различными типами, сначала

```
#l := [1];;
```

и затем

```
#hd(!l) = true;;
```

Но это не приемлемо, поскольку фактически мы могли бы записывать нечто как объект типа `int`, а затем читать как объект типа `bool`. Следовательно, когда используются ссылки, требуются некоторые ограничения на привычные правила let-полиморфизма. Было много предложений по подходящему ограничению системы типов, некоторые из которых весьма запутанные. Недавно различные версии ML похоже, сошлись к относительно простому методу, называемому *ограничением значения* (?), и CAML обеспечивает это ограничение. В самом деле, вышеупомянутая последовательность не выполняется. Но интересно промежуточное поведение. Если мы посмотрим на первую строку, мы увидим:

```
#let l = ref [];;
l : '_a list ref = ref []
```

Подчёркивание перед типом переменной обозначает что `l` не полиморфна в привычном смысле; скорее она имеет один фиксированный тип, хотя тип этот всё ещё неопределён. Эта строка работает отлично:

```
#l := [1];;
- : unit = ()
```

но если посмотреть на тип `l`, то мы увидим:

```
#l;;
- : int list ref = ref [1]
```

Теперь установлен псевдо-полиморфный тип. Благодаря этому, ясно, что следующая строка даст ошибку:

```
#hd(!l) = true;;
Toplevel input:
>hd(!l) = true;;
>          ^^^^^
This expression has type bool,
but is used with type int.
```

Пока это представляется вполне оправданным, но мы ещё не объяснили почему такие типовые переменные с подчёркиванием встречаются и в довольно безобидных чисто функциональных выражениях и, более того, почему они часто исчезают при η -раскрытии, например:

```
#let I x = x;;
I : 'a -> 'a = <fun>
#I o I;;
it : '_a -> '_a = <fun>
#let I2 = I o I in fun x -> I2 x;;
- : '_a -> '_a = <fun>
#fun x -> (I o I) x;;
it : 'a -> 'a = <fun>
```

Другие методы для полиморфных ссылок зачастую зависят от закодированной в типах информации о том, что выражение может содержать ссылки. Это кажется вполне естественным, но такой подход может привести к типам функций захламлённым этой специальной информацией. Представляется непривлекательным, что в типе функции должен отражаться факт того, как функция реализована — императивно или чисто функционально.

Рассмотренное решение, с другой стороны, использует только основной синтаксис let-связанного выражения и перед обобщением типа утверждает, что оно является так называемым *значением*. Чего действительно хотелось бы, так это знать: может ли вычисление выражения привести к побочным эффектам. Но, поскольку в общем случае это неразрешимо, то чтобы выяснить является ли выражение значением или нет, используется простой синтаксический критерий. Грубо говоря, выражение является значением если оно не допускает дальнейших вычислений по правилам ML — вот почему выражение часто может быть обращено в значение, посредством обратного η -преобразования. К сожалению, это работает против методов оптимизации при помощи принудительных вычислений.

Упражнения

1. Определим C комбинатор следующим образом:

```
#let C f x y = f y x;;
```

Что делает эта функция?

```
#fun f l1 l2 -> itlist (union o C map l2 o f) l1 [];;
```

2. Что делает эта функция? Напишите более эффективную версию.

```
#let rec upto n = if n < 1 then [] else append (upto (n-1)) [n];;
```

3. Определим функцию, вычисляющую числа Фибоначчи:

```
#let rec fib =  
  fun 0 -> 0  
    | 1 -> 1  
    | n -> fib(n - 2) + fib(n - 1);;
```

Почему эта функция неэффективна? Предложите лучшую реализацию.

4. Предложите пример использования данного или подобного рекурсивного исключения.

```
#exception Recurse of exn;;
```

5. Напишите простую версию быстрой сортировки на массивах. Сначала массив разделяется на две части по некоторому элементу, затем на левой части и на правой рекурсивно вызывается `sort`. Какой рекурсивный вызов является хвостовым? Сколько требуется памяти в худшем случае? Как с помощью небольшого изменения в коде добиться значительной оптимизации?
6. Докажите, что обе версии `rev`, что мы упомянули, всегда дают одинаковый результат.

Глава 9

Примеры

Как мы говорили, ML изначально был спроектирован как метаязык для автоматизированного доказательства теорем. Однако он пригоден и для многих других приложений, по большей части из области «символьных вычислений». В этой главе мы дадим несколько характерных примеров использования ML. От читателя не требуется досконального знания всех деталей, как, например, в случае приближения вещественных чисел, приведённом далее. Однако важно позапускать эти программы, выполнить упражнения. Поскольку нет лучшего способа почувствовать, как ML используется в действительности.

9.1 Символьное дифференцирование

Фраза «символьное вычисление» довольно неопределённая, но грубо говоря, она охватывает приложения занимающиеся не просто численными расчётами, а манипуляциями над математическими *выражениями*, в общем случае содержащими переменные. Есть несколько успешных «систем компьютерной алгебры» таких как Maxima, Maple и Mathematica. Они могут выполнять полезные математические операции: разложение многочленов на множители, дифференцирование и интегрирование выражений и т.д. Они также пригодны и для обычных вычислений. Мы покажем, как задачи такого рода могут быть решены с помощью ML.

Мы выбрали символьное дифференцирование для нашего примера, потому что для него есть довольно простой и известный алгоритм. Читатель, возможно, знаком с производными основных функций, например $\frac{d}{dx} \sin(x) = \cos(x)$, а также с правилами дифференцирования произведения и композиции функций. Поскольку каждый может систематически использовать эти правила для нахождения производных, то возможно и запрограммировать их на ML.

9.1.1 Термы первого порядка

Мы будем допускать математические выражения в весьма общей форме. Они могут быть построены из переменных и констант, с использованием операторов. Операторы могут быть унарными, бинарными, тернарными, или, в общем случае, арности n . Это выражается с помощью следующего рекурсивного типа данных:

```
#type term = Var of string
           | Const of string
           | Fn of string * (term list);;
Type term defined.
```

Например, выражение:

$$\sin(x + y) / \cos(x - \exp(y)) - \ln(1 + x)$$

представляется:

```
Fn("-", [Fn("/", [Fn("sin", [Fn("+", [Var "x"; Var "y"])]);
                Fn("cos", [Fn("-", [Var "x"; Fn("exp", [Var "y"])])]);
                Fn("ln", [Fn("+", [Const "1"; Var "x"])])])]);;
```

9.1.2 Печать

Чтение и запись выражений в сыром виде довольно неудобны. Это общая проблема для всех систем символьных вычислений, и обычно интерфейс системы содержит *синтаксический анализатор (парсер)* и *систему ввода-вывода*, позволяющую вводить и выводить данные в читаемой форме. Отложим пока подробное обсуждение синтаксического разбора, поскольку эта, сама по себе важная тема, будет обсуждаться далее, а сейчас мы напишем простенькую систему вывода для наших выражений. Чтобы, по крайней мере, можно было видеть что происходит. Мы хотим, чтобы поддерживались обычные соглашения, как-то:

- Переменные и константы записываются по их именам.
- Обычные n -арные функции, выводятся в виде $f(x_1, \dots, x_n)$, т.е. записывается имя функции и рядом, в скобках список аргументов.
- Инфиксные бинарные функции, такие как $+$ записываются между своими аргументами.
- Скобки используются по необходимости.
- В целях сокращения использования скобок, для инфиксных операторов определён приоритет.

Сначала мы объявим список инфиксных операторов – список пар из имени оператора и его приоритета.

```
#let infixes = ["+",10; "-",10; "*",20; "/",20];;
```

Использование списков для представления конечных частичных функций является стандартной практикой. Они обычно называются *ассоциативными списками* и весьма распространены в функциональном программировании.¹ Для того чтобы конвертировать список в частичную функцию мы используем `assoc`:

¹Для более тяжелых приложений больше подойдут альтернативы, такие как хеш-таблицы, но для простых приложений, в которых списки не слишком длинны, решение в виде линейного списка является простым и адекватным.

```
#let rec assoc a ((x,y)::rest) = if a = x then y else assoc a rest;;
Toplevel input:
>let rec assoc a ((x,y)::rest) = if a = x then y else assoc a rest;;
>
Warning: this matching is not exhaustive.
assoc : 'a -> ('a * 'b) list -> 'b = <fun>
```

Компилятор предупреждает нас - вычисление функции может прерваться, если подходящие данные не будут найдены в списке. Теперь мы можем определить функцию, возвращающую приоритет оператора:

```
#let get_precedence s = assoc s infixes;;
get_precedence : string -> int = <fun>
#get_precedence "+";;
- : int = 10
#get_precedence "/";;
- : int = 20
#get_precedence "%";;
Uncaught exception: Match_failure ("", 6544, 6601)
```

Обратите внимание, если мы когда-либо изменим этот список операторов, то всякую функцию, вроде `get_precedence`, использующую его, потребуется переопределить. Это является основной причиной, по которой многие LISP программисты предпочитают динамическое связывание. Множество инфиксных операторов можно сделать произвольно расширяемым, используя обращение по ссылке:

```
#let infixes = ref ["+",10; "-",10; "*",20; "/",20];;
infixes : (string * int) list ref =
  ref ["+", 10; "-", 10; "*", 20; "/", 20]
#let get_precedence s = assoc s (!infixes);;
get_precedence : string -> int = <fun>
#get_precedence "^";;
Uncaught exception: Match_failure ("", 6544, 6601)
#infixes := ("^",30)::(!infixes);;
- : unit = ()
#get_precedence "^";;
- : int = 30
```

Отметим, что по правилам вычисления в ML, разыменование указателя применяется только после того, как `get_precedence` получает свои аргументы, и, следовательно, результаты во множестве операторов. Мы также можем задать функцию, определяющую, является ли некоторая функция инфиксным оператором. Можно просто использовать `get_precedence` и смотреть сработала ли она:

```
#let is_infix s =
  try get_precedence s; true
  with _ -> false;;
```

Альтернатива заключается в использовании общей функции `can`, которая проверяет успешность вычисления другой функции:

```
#let can f x = try f x; true with _ -> false;;
can : ('a -> 'b) -> 'a -> bool = <fun>
#let is_infix = can get_precedence;;
is_infix : string -> bool = <fun>
```

Мы будем использовать следующие функции из уже рассмотренных:

```
#let hd(h::t) = h;;
#let tl(h::t) = t;;
#let rec length l = if l = [] then 0 else 1 + length(tl l);;
```

Без дальнейших хлопот, вот функция для конвертации терма в строку.

```
#let rec string_of_term prec =
  fun (Var s) -> s
  | (Const c) -> c
  | (Fn(f,args)) ->
    if length args = 2 & is_infix f then
      let prec' = get_precedence f in
      let s1 = string_of_term prec' (hd args)
      and s2 = string_of_term prec' (hd(tl args)) in
      let ss = s1^" "^f^" "^s2 in
      if prec' <= prec then "("^ss^")" else ss
    else
      f^"("^(string_of_terms args)^")"

  and string_of_terms t =
    match t with
    [] -> ""
    | [t] -> string_of_term 0 t
    | (h::t) -> (string_of_term 0 h)^","^(string_of_terms t);;
```

Первый аргумент `prec` обозначает уровень приоритета оператора, для которого текущее рассматриваемое выражение является непосредственным подтермом. Так что если текущее выражение имеет инфиксный оператор с некоторым приоритетом, то скобки необходимы, если приоритет этого оператора ниже `prec`. Например, если мы рассматриваем правую часть $x * (y + z)$. На самом деле, для более наглядной группировки мы используем скобки даже когда `prec` и приоритет текущего оператора равны. В случае ассоциативных операторов, вроде $+$ это не важно, но мы должны различать $x - (y - z)$ и $(x - y) - z$. (Более усложнённая схема могла бы учитывать ассоциативность каждого оператора.) Вторая, взаимно рекурсивная, функция `string_of_terms` используется для печати списка термов через запятую, как они идут в неинфиксной форме $f(t_1, \dots, t_n)$. Давайте посмотрим это в действии:

```
#let t =
  Fn("-", [Fn("/", [Fn("sin", [Fn("+", [Var "x"; Var "y"])]);
                    Fn("cos", [Fn("-", [Var "x";
                                      Fn("exp", [Var "y"])])])]);
          Fn("ln", [Fn("+", [Const "1"; Var "x"])])]);
t : term =
  Fn
    ("-",
     [Fn
       ("/",
        [Fn ("sin", [Fn ("+", [Var "x"; Var "y"])]);
          Fn ("cos", [Fn ("-", [Var "x"; Fn ("exp", [Var "y"])])]);
          Fn ("ln", [Fn ("+", [Const "1"; Var "x"])])])]
#string_of_term 0 t;;
- : string = "sin(x + y) / cos(x - exp(y)) - ln(1 + x)"
```

На самом деле, нам даже не требуется самостоятельно конвертировать терм в строку. CAML Light позволяет нам настроить свою систему вывода. Это требует несколько специальных команд:

```
##open "format";;
#let print_term s =
  open_hvbox 0;
  print_string("^(string_of_term 0 s)^");
  close_box();;
print_term : term -> unit = <fun>
#install_printer "print_term";;
- : unit = ()
```

Почувствуйте разницу:

```
#let t =
  Fn("-", [Fn("/", [Fn("sin", [Fn("+", [Var "x"; Var "y"])]);
                    Fn("cos", [Fn("-", [Var "x";
                                      Fn("exp", [Var "y"])])])]);
          Fn("ln", [Fn("+", [Const "1"; Var "x"])])]);
t : term = 'sin(x + y) / cos(x - exp(y)) - ln(1 + x)'
#let x = t
x : term = 'sin(x + y) / cos(x - exp(y)) - ln(1 + x)'
```

После того как новый принтер установлен, он будет использоваться всякий раз при печати типа `term`, даже если он представляет собой составной тип данных, как, например, кортеж:

```
##(x,t);;
- : term * term =
  'sin(x + y) / cos(x - exp(y)) - ln(1 + x)',
  'sin(x + y) / cos(x - exp(y)) - ln(1 + x)'
```

или список:

```
#[x; t; x];;
- : term list =
  ['sin(x + y) / cos(x - exp(y)) - ln(1 + x)';
   'sin(x + y) / cos(x - exp(y)) - ln(1 + x)';
   'sin(x + y) / cos(x - exp(y)) - ln(1 + x)']
```

Подобная печать всё же довольно груба, поскольку большие выражения не разбиваются аккуратно на линии. Библиотека `format`, из которой мы использовали несколько функций выше, предлагает лучшее решение. Вместо конвертации терма в строку и печати, мы можем выполнить отдельные вызовы печати для каждой его части, с помощью специальной функции, помогающей системе печати разбивать строки. Многие из принципов, использующиеся в этой и подобных системах форматированного вывода рассматриваются в Oppen (1980). Мы не будем рассматривать их в книге, за подробностями обращайтесь к руководству по CAML.²

9.1.3 Дифференцирование

Перейдём теперь к довольно простой задаче нахождения производной от функции. Сначала давайте вспомним, чему нас обучали в школе:

- Если выражение - одна из элементарных функций дифференцируемого аргумента, например $\sin(x)$, то результат дифференцирования – известная производная этой функции.
- Если выражение представимо в форме $f(x) + g(x)$, то по правилу суммы, результат дифференцирования есть $f'(x) + g'(x)$. Аналогично для разности.
- Если выражение задано в форме $f(x) * g(x)$, применяется *правило Лейбница*, т.е. возвращается $f'(x) * g(x) + f(x) * g'(x)$.
- Если выражение является одной из композицией стандартных функций: $f(g(x))$, то применяется правило последовательного дифференцирования, результат которого: $g'(x) * f'(g(x))$.

Таким образом, эти правила представляют собой рекурсивный алгоритм, хотя такая терминология редко используется в школах. Мы можем дословно запрограммировать его в ML:

²Там также есть учебник на эту тему, написанный Pierre Weis. Адрес: <http://caml.inria.fr/index-eng.html>

```
#let rec differentiate x tm = match tm with
  Var y -> if y = x then Const "1" else Const "0"
| Const c -> Const "0"
| Fn("-", [t]) -> Fn("-", [differentiate x t])
| Fn("+", [t1; t2]) -> Fn("+", [differentiate x t1;
                                differentiate x t2])
| Fn("-", [t1; t2]) -> Fn("-", [differentiate x t1;
                                differentiate x t2])
| Fn("*", [t1; t2]) ->
  Fn("+", [Fn("*", [differentiate x t1; t2]);
           Fn("*", [t1; differentiate x t2])])
| Fn("inv", [t]) -> chain x t
  (Fn("-", [Fn("inv", [Fn("^", [t; Const "2"])])]))
| Fn("^", [t; n]) -> chain x t
  (Fn("*", [n; Fn("^", [t; Fn("-", [n; Const "1"])])]))
| Fn("exp", [t]) -> chain x t tm
| Fn("ln", [t]) -> chain x t (Fn("inv", [t]))
| Fn("sin", [t]) -> chain x t (Fn("cos", [t]))
| Fn("cos", [t]) -> chain x t
  (Fn("-", [Fn("sin", [t])]))
| Fn("/", [t1; t2]) -> differentiate x
  (Fn("*", [t1; Fn("inv", [t2])]))
| Fn("tan", [t]) -> differentiate x
  (Fn("/", [Fn("sin", [t]); Fn("cos", [t])]))
and chain x t u = Fn("*", [differentiate x t; u]);;
```

Функция `chain` предназначена просто для упрощения последовательного дифференцирования. Мы также систематически используем правила для сумм, произведений и пр. а также производные стандартных функций. Конечно, мы могли бы, по желанию, добавить и другие функции, например, гиперболические или обратные тригонометрические. В паре случаев, а именно, для деления и функции *tan*, мы избегаем усложнённого определения за счёт альтернативной формулировки.

9.1.4 Упрощение

Если мы опробуем `differentiate` на нашем рабочем примере, она будет работать довольно хорошо:

```
#t;;
- : term = 'sin(x + y) / cos(x - exp(y)) - ln(1 + x)'
#differentiate "x" t;;
- : term =
  '(((1 + 0) * cos(x + y)) * inv(cos(x - exp(y))) +
    sin(x + y) * (((1 - 0 * exp(y)) * -(sin(x - exp(y)))) *
      -(inv(cos(x - exp(y)) ^ 2)))) - (0 + 1) * inv(1 + x)'
#differentiate "y" t;;
- : term =
  '(((0 + 1) * cos(x + y)) * inv(cos(x - exp(y))) +
    sin(x + y) * (((0 - 1 * exp(y)) * -(sin(x - exp(y)))) *
      -(inv(cos(x - exp(y)) ^ 2)))) - (0 + 0) * inv(1 + x)'
#differentiate "z" t;;
- : term =
  '(((0 + 0) * cos(x + y)) * inv(cos(x - exp(y))) +
    sin(x + y) * (((0 - 0 * exp(y)) * -(sin(x - exp(y)))) *
      -(inv(cos(x - exp(y)) ^ 2)))) - (0 + 0) * inv(1 + x)'
```

Однако она не выполняет различные очевидные упрощения, такие как $0 * x = 0$ и $x + 0 = x$. Некоторые из этих избыточных выражений создаются нашей прямолинейной стратегией дифференцирования, которая, например, применяет правило последовательного дифференцирования $f(t)$ даже когда t – всего лишь переменная дифференцирования. Вместо избыточного усложнения функции дифференцирования давайте зададим отдельную функцию упрощения, которая будет избавляться от необязательных выражений.

```
#let simp =
  fun (Fn("+",[Const "0"; t])) -> t
  | (Fn("+",[t; Const "0"])) -> t
  | (Fn("-",[t; Const "0"])) -> t
  | (Fn("-",[Const "0"; t])) -> Fn("-",[t])
  | (Fn("+",[t1; Fn("-",[t2])])) -> Fn("-",[t1; t2])
  | (Fn("*",[Const "0"; t])) -> Const "0"
  | (Fn("*",[t; Const "0"])) -> Const "0"
  | (Fn("*",[Const "1"; t])) -> t
  | (Fn("*",[t; Const "1"])) -> t
  | (Fn("*",[Fn("-",[t1]); Fn("-",[t2])])) -> Fn("*",[t1; t2])
  | (Fn("*",[Fn("-",[t1]); t2])) -> Fn("-",[Fn("*",[t1; t2])])
  | (Fn("*",[t1; Fn("-",[t2])])) -> Fn("-",[Fn("*",[t1; t2])])
  | (Fn("-",[Fn("-",[t])])) -> t
  | t -> t;;
```

Упрощение применяется, начиная с корня дерева выражения. Мы выполняем его в обходе снизу - вверх. В некоторых случаях было бы более эффективно упрощение при обходе сверху - вниз, например, для $0 * t$ и сложного выражения t , но это потребует повторения в случае термов вроде $(0+0)*2$. Выбор здесь напоминает выбор между нормальным и аппликативным порядком редукции в лямбда-исчислении.

```
#let rec dsimp =
  fun (Fn(fn,args)) -> simp(Fn(fn,map dsimp args))
  | t -> simp t;;
```

Теперь мы получаем лучшие результаты:

```
#dsimp(differentiate "x" t);;
- : term =
  '(cos(x + y) * inv(cos(x - exp(y))) +
    sin(x + y) * (sin(x - exp(y)) *
      inv(cos(x - exp(y)) ^ 2))) - inv(1 + x)'
#dsimp(differentiate "y" t);;
- : term =
  'cos(x + y) * inv(cos(x - exp(y))) -
    sin(x + y) * ((exp(y) * sin(x - exp(y))) *
      inv(cos(x - exp(y)) ^ 2))'
#dsimp(differentiate "z" t);;
- : term = '0'
```

В общем, всегда можно добавить более усложнённые правила упрощения. Например, рассмотрим:


```
#let t2 = Fn("tan",[Var "x"]);;
t2 : term = 'tan(x)'
#differentiate "x" t2;;
- : term =
  '(1 * cos(x)) * inv(cos(x)) +
   sin(x) * ((1 * -(sin(x))) * -(inv(cos(x) ^ 2)))'
#dsimp(differentiate "x" t2);;
- : term = 'cos(x) * inv(cos(x)) +
           sin(x) * (sin(x) * inv(cos(x) ^ 2))'
```

Мы хотим упростить $\cos(x) * \text{inv}(\cos(x))$ до 1, пренебрегая, как это делает большинство коммерческих систем компьютерной алгебры, возможностью обращения $\cos(x)$ в нуль. Мы, конечно, можем добавить дополнительные правила упрощения. В то же время, можно было бы сгруппировать множители во втором слагаемом. И здесь, мы начинаем понимать, имеет значение человеческая психология. Что предпочтительней? Для машины самостоятельное решение будет затруднительно.

```
sin(x) * (sin(x) * inv(cos(x) ^ 2))

sin(x) ^ 2 * inv(cos(x) ^ 2)

sin(x) ^ 2 / cos(x) ^ 2

(sin(x) / cos(x)) ^ 2

tan(x) ^ 2
```

Выбрав второй вариант, затем следует заменить

```
1 + tan(x) ^ 2
```

на

```
sec(x) ^ 2
```

Такая запись и короче, и более привычна. Тем не менее, такое решение основывается на том, что мы знаем определения довольно редко используемых тригонометрических функций, вроде \sec . Что бы машина не решила делать, вероятно, найдётся пользователь, которого это не устроит.

9.2 Синтаксический анализ

Недостатком предыдущего примера является необходимость задавать входные данные в виде композиции конструкторов типов. В этом разделе мы рассмотрим задачу построения формального языка описания выражений и разработки его синтаксического анализатора. Изложенный подход будет при этом достаточно общим, чтобы впоследствии легко применяться к аналогичным языкам.

Сформулируем задачу синтаксического анализа в общем виде. Формальный язык определяется своей *грамматикой*, которая задает множество *продукций* (*правил вывода*) для каждой синтаксической категории. Для языка термов, включающих

лишь две инфиксные операции, $+$ и $*$, а также числовые константы $(0, 1, \dots)$ и переменные с алфавитно-цифровыми именами, грамматика может выглядеть так:

$$\begin{aligned}
 term &\longrightarrow name(termlist) \\
 &\quad | \quad name \\
 &\quad | \quad (term) \\
 &\quad | \quad numeral \\
 &\quad | \quad -term \\
 &\quad | \quad term + term \\
 &\quad | \quad term * term \\
 termlist &\longrightarrow term, termlist \\
 &\quad | \quad term
 \end{aligned}$$

На данный момент будем предполагать, что нам заранее известны множества допустимых имен и числовых констант, но мы могли бы также определить их при помощи аналогичных правил на основе символов входного алфавита. Заданное множество продукций предоставляет нам способ порождения конкретного линейного представления всевозможных термов. Отметим, что имена вида *name*, *numeral*, *term* и *termlist* обозначают синтаксические *метанпеременные*, а набранные полужирным шрифтом символы «+», «(» и так далее — принадлежат входному языку. Рассмотрим пример применения правил вывода:

$$\begin{aligned}
 term &\longrightarrow term * term \\
 &\longrightarrow term * (term) \\
 &\longrightarrow term * (term + term) \\
 &\longrightarrow term * (term + term * term) \\
 &\longrightarrow numeral * (name + name * name)
 \end{aligned}$$

откуда, подставив вместо *name* и *numeral* некоторые значения из соответствующих множеств, мы можем породить такую строку:

$$10 * (x + y * z)$$

Задача *синтаксического анализа (разбора)* преследует прямо противоположную цель — восстановить последовательность применения правил вывода, то есть по заданной строке определить, каким образом она могла быть порождена. Как правило, результатом анализа является *дерево разбора*, наглядно демонстрирующее порядок выбора продукций.

Одной из проблем, возникающей в ходе синтаксического анализа, является *неоднозначность* грамматики, т. е. возможность порождения заданной строки несколькими различными способами. В рассмотренном выше примере та же самая строка могла быть порождена иным путём:

$$\begin{aligned}
 term &\longrightarrow term * term \\
 &\longrightarrow term * (term) \\
 &\longrightarrow term * (term * term) \\
 &\longrightarrow term * (term + term * term) \\
 &\longrightarrow numeral * (name + name * name)
 \end{aligned}$$

Очевидная возможность устранения неоднозначности — назначение инфиксным операциям приоритетов и ассоциативности. Кроме того, мы можем добиться такого же эффекта без привлечения дополнительных механизмов за счёт добавления новых синтаксических категорий:

$$\begin{array}{lcl}
 atom & \longrightarrow & name(termlist) \\
 & | & name \\
 & | & numeral \\
 & | & (term) \\
 & | & -atom \\
 mulexp & \longrightarrow & atom * mulexp \\
 & | & atom \\
 term & \longrightarrow & mulexp + term \\
 & | & mulexp \\
 termlist & \longrightarrow & term, termlist \\
 & | & term
 \end{array}$$

Отметим, что такая модификация грамматики делает обе инфиксные операции правоассоциативными. Более сложным примером использования этого приёма служит формальное определение понятия «выражение» в стандарте ANSI C.

9.2.1 Метод рекурсивного спуска

Строение правил вывода подсказывает очень простой алгоритм синтаксического анализа при помощи множества взаимно рекурсивных функций. Его суть в том, что для каждой синтаксической категории вводится своя функция, причем структура рекурсивных вызовов соответствует зависимости одних продукций от других. Например, процедура анализа термов `term` при получении из входной последовательности символа — будет рекурсивно вызывать саму себя для анализа аргумента операции, а при встрече имени, за которым следует открывающая скобка, обратится к процедуре `termlist`. Последняя, в свою очередь, вызовет `term` не менее одного раза, и так далее. Такой подход естественно выражается на языке, подобном ML, для которого рекурсия служит одной из основных управляющих конструкций.

Предположим, что наш синтаксический анализатор, реализованный средствами ML, получает в качестве входной последовательности список объектов некоторого типа α . Возможно, чтобы входная последовательность состояла из простых символов, но на практике обычно вводится дополнительный уровень *лексического анализа*, в ходе которого символы группируются в *лексемы (токены)*, например, «x12», «:=» и «96», так что входная последовательность состоит уже из лексем, о типе которых мы не будем делать предположений. Аналогично, будем считать, что анализатор производит результат типа β . Этот результат может быть, например, деревом разбора, представленным рекурсивным типом данных, либо просто числом, если анализатор предназначен для разбора выражения и его вычисления одновременно. В общем случае, анализатор может не обработать входную последовательность полностью, так что нам потребуется вернуть в

дополнение к результату разбора еще и список необработанных лексем. Таким образом, тип анализатора:

$$(\alpha)list \rightarrow \beta \times (\alpha)list$$

Например, для входной последовательности $(x + y) * z$ функция `atom` должна обработать $(x + y)$ и вернуть $* z$. Результирующее дерево разбора принятой части входа может быть представлено значением рекурсивного типа, введенного ранее. Отсюда мы имеем

```
atom "(x_+_y)_*_z" = Fn("+" , [Var "x" ; Var "y" ] ) , "*_z"
```

Поскольку любое обращение к функции `atom` должно происходить из функции `mulexp`, то последняя будет использовать результат, полученный вычислением `atom`, а также обрабатывать оставшиеся лексемы, вызывая повторно `atom` для анализа подвыражения z .

9.2.2 Комбинаторы синтаксического анализа

Еще одна причина, по которой предложенный метод разбора особенно хорошо подходит для реализации на языке ML, это возможность определения некоторых полезных комбинаторов, при помощи которых новые анализаторы легко создаются на базе уже существующих. Определив эти комбинаторы как инфиксные операции, мы в состоянии придать программе синтаксического анализатора вид, очень схожий со структурой исходной грамматики.

Для начала введём исключение, сигнализирующее об ошибках анализа. Далее определим инфиксную операцию `++`, которая применяет два анализатора последовательно, объединяя их результаты, а также инфиксную операцию `||`, которая вначале делает попытку применить один из анализаторов, а затем — второй. Операция `many` представляет собой свертку относительно `++`, т. е. применяет заданный анализатор максимально возможное количество раз, выдавая список результатов. Наконец, инфиксная операция `>>` применяется для завершающей обработки результатов анализа заданной функцией.

Согласно синтаксису CAML, идентификаторы наподобие `++` автоматически считаются инфиксными операциями, поэтому определение операций включает временную блокировку этой возможности при помощи ключевого слова `prefix`. Приоритеты также задаются автоматически по первым символам идентификаторов и полагаются равными приоритетам арифметических операций, обозначенных этими символами. Таким образом, приоритет `++` наивысший, `>>` — средний, `||` — низший, что нам и требуется.

```

exception Noparse;;

let prefix || parser1 parser2 input =
  try parser1 input
  with Noparse -> parser2 input;;

let prefix ++ parser1 parser2 input =
  let result1,rest1 = parser1 input in
  let result2,rest2 = parser2 rest1 in
  (result1,result2),rest2;;

let rec many parser input =
  try let result,next = parser input in
      let results,rest = many parser next in
      (result::results),rest
  with Noparse -> [],input;;

let prefix >> parser treatment input =
  let result,rest = parser input in
  treatment(result),rest;;

```

Введём следующие универсальные функции, которые нам понадобятся в дальнейшем. Большая часть из них уже обсуждалась ранее, за исключением функции `explode`, преобразующей строку в список односимвольных строк. Её реализация использует встроенные функции `sub_string` и `string_length`, которые не рассматривались, но их назначение легко понять из примера.

```

let rec itlist f =
  fun [] b -> b
  | (h::t) b -> f h (itlist f t b);;

let uncurry f(x,y) = f x y;;

let K x y = x;;

let C f x y = f y x;;

let o f g x = f(g x);;
#infix "o";;

let explode s =
  let rec exap n l =
    if n < 0 then l else
    exap (n - 1) ((sub_string s n 1)::l) in
  exap (string_length s - 1) [];;

```

Для начала, определим некоторые «атомарные» анализаторы. Функция `some` принимает любой входной символ, удовлетворяющий заданному предикату, и возвращает его. Функция `a` выполняет схожее действие, с той разницей, что она проверяет входной символ на равенство заданному. Наконец, `finished` предназначена для проверки того, что вся входная последовательность была обработана.

```

let some p =
  fun [] -> raise Noparse
    | (h::t) -> if p h then (h,t) else raise Noparse;;

let a tok = some (fun item -> item = tok);;

let finished input =
  if input = [] then 0,input else raise Noparse;;

```

9.2.3 Лексический анализ

Комбинаторы синтаксического анализа в сочетании с несколькими простыми функциями классификации символов хорошо подходят для построения лексического анализатора нашего языка термов. Прежде всего, определим тип, представляющий лексемы (*токены*), после чего реализуем лексический анализатор, преобразующий входную последовательность в список лексем. Лексической категории *Other* соответствуют обозначения операций и т. п., причем в нашем случае все они весьма просты и состоят лишь из одного символа (в отличие от составных обозначений, таких как `:=`).

```

type token = Name of string | Num of string | Other of string;;

let lex =
  let several p = many (some p) in
  let lowercase_letter s = "a" <= s & s <= "z" in
  let uppercase_letter s = "A" <= s & s <= "Z" in
  let letter s = lowercase_letter s or uppercase_letter s in
  let alpha s = letter s or s = "_" or s = "'" in
  let digit s = "0" <= s & s <= "9" in
  let alphanum s = alpha s or digit s in
  let space s = s = " " or s = "\n" or s = "\t" in
  let collect(h,t) = h^(itlist (prefix ^) t "") in
  let rawname =
    some alpha ++ several alphanum >> (Name o collect) in
  let rawnumeral =
    some digit ++ several digit >> (Num o collect) in
  let rawother = some (K true) >> Other in
  let token =
    (rawname || rawnumeral || rawother) ++ several space >> fst in
  let tokens = (several space ++ many token) >> snd in
  let alltokens = (tokens ++ finished) >> fst in
  fst o alltokens o explode;;

```

Например,

```
#lex "sin(x + y) * cos(2 * x + y)";  
- : token list =  
[Name "sin"; Other "("; Name "x"; Other "+"; Name "y"; Other ")";  
 Other "*"; Name "cos"; Other "("; Num "2"; Other "*"; Name "x";  
 Other "+"; Name "y"; Other ")"]
```

9.2.4 Анализатор термов

Для завершения перехода от анализа отдельных символов к анализу лексем, введём базовые анализаторы, принимающие лексемы заданной категории:

```
let name =  
  fun (Name s::rest) -> s,rest  
  | _ -> raise Noparse;;  
  
let numeral =  
  fun (Num s::rest) -> s,rest  
  | _ -> raise Noparse;;  
  
let other =  
  fun (Other s::rest) -> s,rest  
  | _ -> raise Noparse;;
```

С помощью этих функций мы можем определить анализатор термов в виде, очень схожем с исходной грамматикой. Основное различие состоит в том, что каждой продукции сопоставлено некоторое действие, результат которого возвращается как результат анализа.

```

let rec atom input
  = (name ++ a (Other "(") ++ termlist ++ a (Other ")")
    >> (fun ((name, _), args), _) -> Fn(name, args))
  || name
    >> (fun s -> Var s)
  || numeral
    >> (fun s -> Const s)
  || a (Other "(") ++ term ++ a (Other ")")
    >> (snd o fst)
  || a (Other "-") ++ atom
    >> snd) input
and mulexp input
  = (atom ++ a (Other "*") ++ mulexp
    >> (fun ((a, _), m) -> Fn("*", [a; m])))
  || atom) input
and term input
  = (mulexp ++ a (Other "+") ++ term
    >> (fun ((a, _), m) -> Fn("+", [a; m])))
  || mulexp) input
and termlist input
  = (term ++ a (Other ",") ++ termlist
    >> (fun ((h, _), t) -> h :: t))
  || term
    >> (fun h -> [h])) input;;

```

Объединим определённые ранее примитивы в единую функцию:

```

let parser = fst o (term ++ finished >> fst) o lex;;

```

Наглядной иллюстрацией работы этой функции является её вызов до и после установки специализированной функции вывода (см. выше):

```

#parser "sin(x + y) * cos(2 * x + y)";;
- : term =
  Fn
    ("*",
      [Fn ("sin", [Fn ("+", [Var "x"; Var "y"])]);
       Fn ("cos", [Fn ("+", [Fn ("*", [Const "2"; Var "x"]);
                               Var "y"])])]))
#install_printer "print_term";;
- : unit = ()
#parser "sin(x + y) * cos(2 * x + y)";;
- : term = 'sin(x + y) * cos(2 * x + y)'

```

9.2.5 Автоматический учёт приоритетов

Синтаксический анализатор, рассмотренный выше, реализует разбор двух инфиксных операций в явном виде. Однако, для большего удобства и согласованности с нашим подходом к выводу выражений, следует учесть введенную

ранее информацию о множестве доступных инфиксных операций. Более того, наличие даже двух различных бинарных операций заставляет прибегать к искусственным приёмам при построении грамматики и программы разбора — нетрудно представить себе, во что превратится работа, например, с дюжиной таких операций. Таким образом, несомненна полезность автоматического порождения иерархии анализаторов для каждой доступной операции, упорядоченных согласно их приоритетам. Решение этой задачи достаточно просто. Прежде всего, зададим обобщённую функцию, которая принимает в качестве аргумента анализатор синтаксической категории, которая на данном уровне приоритета считается атомарным выражением, и возвращает новый анализатор, предназначенный для разбора последовательности таких атомарных выражений, объединённых заданной операцией. Заметим, что того же эффекта мы можем добиться при помощи нашего набора комбинаторов, но функция, приведенная ниже, проще и эффективнее.

```
let rec binop op parser input =
  let atom1, rest1 as result = parser input in
  if not rest1 = [] & hd rest1 = Other op then
    let atom2, rest2 = binop op parser (tl rest1) in
    Fn(op, [atom1; atom2]), rest2
  else result;;
```

Далее определим функции, которые извлекают из нашего ассоциативного списка очередную операцию с наименьшим (равным предыдущему) приоритетом. Если список был заранее сортирован по приоритетам, мы можем просто выбирать очередной элемент по порядку следования.

```
let findmin l = itlist
  (fun (_, pr1 as p1) (_, pr2 as p2) -> if pr1 <= pr2 then p1 else p2)
  (tl l) (hd l);;

let rec delete x (h::t) = if h = x then t else h::(delete x t);;
```

Обобщенный анализатор множества бинарных операций с приоритетами:

```
let rec precedence ilist parser input =
  if ilist = [] then parser input else
  let opp = findmin ilist in
  let ilist' = delete opp ilist in
  binop (fst opp) (precedence ilist' parser) input;;
```

Использование этого вспомогательного анализатора позволяет упростить основной и сделать его более общим:

```

let rec atom input
  = (name ++ a (Other "(") ++ termlist ++ a (Other ")")
    >> (fun ((name, _), args), _) -> Fn(name, args))
  || name
    >> (fun s -> Var s)
  || numeral
    >> (fun s -> Const s)
  || a (Other "(") ++ term ++ a (Other ")")
    >> (snd o fst)
  || a (Other "-") ++ atom
    >> snd) input
and term input = precedence (!infixes) atom input
and termlist input
  = (term ++ a (Other ",") ++ termlist
    >> (fun ((h, _), t) -> h :: t)
  || term
    >> (fun h -> [h])) input;;

let parser = fst o (term ++ finished >> fst) o lex;;

```

Пример разбора выражения:

```

#parser "2 * sin(x)^2 + 2 * sin(y)^2 - 2";;
- : term =
  Fn
    ("+",
      [Fn ("*", [Const "2"; Fn ("^", [Fn ("sin", [Var "x"]);
                                         Const "2"])]);
      Fn
        ("-",
          [Fn ("*", [Const "2"; Fn ("^", [Fn ("sin", [Var "y"]);
                                         Const "2"])]);
          Const "2"])]))
#install_printer "print_term";;
- : unit = ()
#parser "2 * sin(x)^2 + 2 * sin(y)^2 - 2";;
- : term = '2 * sin(x) ^ 2 + (2 * sin(y) ^ 2 - 2)'

```

9.2.6 Недостатки метода

Наш подход к лексическому и синтаксическому анализу не отличается особой эффективностью. Действительно, CAML и некоторые другие реализации языка ML включают генератор синтаксических анализаторов для LR-грамматик, аналогичный популярной в Unix-среде системе YACC (Yet Another Compiler Compiler) для языка C. Эти генераторы не только охватывают более широкий класс грамматик, но и производят более эффективные анализаторы. Однако, мы считаем, что предложенный метод достаточно прозрачен, приемлем в большинстве случаев и является хорошим введением в программирование с использованием функций высших порядков.

Небольшая доработка некоторых особенно ресурсоёмких фрагментов анализатора может заметно улучшить его общую эффективность. Так, при

наличии в одной синтаксической категории нескольких продукций с общим префиксом, следует избегать его многократного анализа, который может оказаться нетривиальным. Примером подобной ситуации служат правила вывода для термов:

$$\begin{array}{lcl} term & \longrightarrow & name(termlist) \\ & | & name \\ & | & \dots \end{array}$$

Мы намеренно поместили в ходе реализации более длинное правило первым, так как в противном случае успешное чтение имени приведёт в дальнейшем к отказу при попытке разбора списка аргументов. Однако, возникающее при этом повторное сканирование избыточно. В данном случае цена избыточности невелика, поскольку повторному анализу подвергается лишь одна лексема, но потенциальные накладные расходы при разборе *termlist* могут оказаться более существенными:

```
let ...
and termlist input
  = (term ++ a (Other ",") ++ termlist
    >> (fun ((h,_),t) -> h::t)
    || term
    >> (fun h -> [h])) input;;
```

В отличие от предыдущего примера, здесь повторному сканированию в случае возврата подвергается целый терм, который может быть сколь угодно сложным. Существуют различные способы избежать этого. Например, мы можем преобразовать правила вывода так, чтобы выбор одного из них происходил уже после того, как прочитан начальный терм. Комбинатор **many** позволяет легко устранить явную рекурсию:

```
let ...
and termlist input
  = term ++ many (a (Other ",") ++ term >> snd)
    >> (fun (h,t) -> h::t) input;;
```

Таким образом, итоговая реализация анализатора имеет вид:

```

let rec atom input
  = (name ++ a (Other "(") ++ termlist ++ a (Other ")")
    >> (fun ((name, _), args), _) -> Fn(name, args))
  || name
    >> (fun s -> Var s)
  || numeral
    >> (fun s -> Const s)
  || a (Other "(") ++ term ++ a (Other ")")
    >> (snd o fst)
  || a (Other "-") ++ atom
    >> snd) input
and term input = precedence (!infixes) atom input
and termlist input
  = (term ++ many (a (Other ",") ++ term >> snd)
    >> (fun (h, t) -> h :: t)) input;;

```

Общим недостатком нашего подхода и метода рекурсивного спуска в целом являются проблемы с разбором *леворекурсивных* грамматик. Правило вывода некоторой синтаксической категории называется леворекурсивным, если его правая часть начинается с этой же категории. Например, если бы мы попытались определить в нашей грамматике левоассоциативную операцию сложения, это можно было бы выразить так:

$$\begin{array}{l}
 term \longrightarrow term + mulexp \\
 \quad \quad | \quad mulexp
 \end{array}$$

Прямое отображение этих правил вывода на язык ML приведет к заикливанию, поскольку функция `term` будет вызываться для одного и того же подвыражения снова и снова. Существуют различные, причём не слишком универсальные, способы решения проблемы. Например, мы можем реализовать в явном виде цикл вместо рекурсии, применение которой в данном контексте не имеет под собой особых оснований. Для этого воспользуемся стандартными комбинаторами наподобие `many`, чтобы получить список деревьев разбора подвыражений вида *mulexp*, после чего построим по этому списку результирующее левоассоциативное дерево.

В заключение отметим, что наша обработка ошибок разбора не отличается особой элегантностью. При возникновении любого затруднения мы порождаем исключение `Noparse`, обработка которого представляет собой возврат. Такое поведение, однако, не всегда подходит для типичных грамматик. В лучшем случае, оно приведёт к интенсивному повторному сканированию, но может вызвать и последовательность возвратов вплоть до корня дерева разбора. Например, встретив в ходе анализа подвыражение `5 +` мы ожидаем, что вслед за символом `+` следует очередной терм. Если он не был найден, следует выдать пользователю внятное сообщение об ошибке вместо простой генерации `Noparse` и, возможно, последующих безуспешных попыток разобрать выражение другим путём.

9.3 Точная арифметика вещественных чисел

Машинная реализация вещественной арифметики обычно использует приближённое представление чисел в формате с плавающей точкой. В общем случае, мы можем оперировать вещественными числами (либо вручную, либо при помощи компьютера) лишь в том случае, когда они имеют то или иное конечное представление. Возникает вопрос, уместно ли говорить о ‘существовании’ чисел, которые не представимы в конечной форме. Например, Кронекер признавал целые и рациональные числа, поскольку их можно задать точно, а также *алгебраические* числа³, представимые многочленами, корнями которых они являются. Однако, он отвергал трансцендентные числа, как не имеющие конечного представления. Говорят, что когда Кронекеру попалось на глаза известное доказательство трансцендентности числа π Lindemann (1882), он якобы отозвался о нем так: «интересно, если не принимать во внимание того, что числа π не существуют».

Учитывая современные достижения, мы можем сказать, что конечное представление в принципе возможно для гораздо большего количества чисел, чем те, существование которых признавалось Кронекером. Этим представлением являются программы (или, вообще говоря, *правила*) вычисления требуемых чисел с произвольной заданной разрядностью. Например, мы можем написать программу, вычисляющую для заданного n первые n знаков числа π или же рациональное число r такое, что $|\pi - r| < 2^{-n}$. Независимо от того, какой из подходов был выбран для последовательного уточнения вещественного числа, важнейшим его свойством является конечность программы-представления.

Такой подход особенно хорошо реализуется на языках, подобных ML, при помощи функций высших порядков. То, что мы называли выше «программой», в ML-реализации становится просто функцией. Арифметические операции при этом представимы функциями высших порядков, которые по заданным аппроксимациям x и y конструируют аппроксимации для $x + y$, xy , $\sin(x)$ и т.д. В более традиционных языках программирования нам пришлось бы задать конкретное представление программ, например, гёделеву нумерацию, и реализовать для него интерпретатор.⁴

9.3.1 Выбор представления вещественных чисел

Каждому вещественному числу x поставим в соответствие функцию $f_x : \mathbb{N} \rightarrow \mathbb{Z}$, которая для произвольного $n \in \mathbb{N}$ возвращает масштабированное приближённое значение x с точностью 2^{-n} :

$$|f_x(n) - 2^n x| < 1.$$

В свою очередь, данное выражение эквивалентно $|\frac{f_x(n)}{2^n} - x| < \frac{1}{2^n}$. Также возможно непосредственное вычисление рационального приближённого значения, но для удобства вычислений важно, чтобы знаменатели всех вычисляемых величин были представимы в виде степеней некоторого числа. Подобное ограничение позволит избежать лавинообразного роста общего знаменателя при суммировании. Явное масштабирование упрощает реализацию операций, позволяя ограничиться

³Вещественное число называется алгебраическим, если оно является корнем полинома с целыми коэффициентами (например, $\sqrt{2}$), в противном случае это число — трансцендентное.

⁴Effectively, we are manipulating functions ‘in extension’ rather than ‘in intension’, i.e. not concerning ourselves with their representation.

лишь целочисленной арифметикой. Выбор 2 в качестве основания произволен. Малые основания предпочтительнее тем, что позволяют подбирать подходящую точность более гибко. Например, выбор основания 10 ведёт к тому, что даже при необходимости небольшого увеличения точности представления мы вынуждены увеличивать её в 10 раз.

9.3.2 Целые числа произвольной разрядности

Стандартный целочисленный тип (`int`) в CAML имеет довольно ограниченный диапазон представимых значений, поэтому нам прежде всего потребуется возможность оперировать целыми числами неограниченной разрядности. Программная реализация подобной арифметики не слишком сложна, но несколько утомительна. К счастью, версия CAML Light, установленная на Thor, уже включает в себя библиотеку быстрых алгоритмов целочисленной (на самом деле, рациональной) арифметики. Предположим, что конфигурация системы задает следующие пути поиска исполнимых файлов:

```
PATH="$PATH:/home/jrh13/caml/bin"
export PATH
```

В этом случае CAML-система с поддержкой «длинной» арифметики запускается командой:

```
$ camllight my_little_caml
```

После запуска системы используйте директиву `#open` для получения доступа ко всем функциям библиотеки:

```
##open "num";;
```

В библиотеке определяется новый тип данных `num`, представляющий рациональные числа произвольной разрядности, среди которых нам понадобится лишь подмножество целых чисел. Язык CAML не предоставляет возможности перегрузки операций, поэтому для обозначения арифметических действий над `num` приходится использовать другие символы.

Отметим, что числа-константы типа `num` должны задаваться как `Int k`, а не просто `k`. Фактически, `Int` является конструктором типа `num` и применяется в случае, когда число имеет прямое машинное представление. Большие числа конструируются `Big_int`.

Унарная операция смены знака величины типа `num` обозначается `minus_num`. Ещё одна полезная унарная операция — `abs_num`, которая вычисляет абсолютную величину аргумента, т. е. по заданному x возвращает $|x|$.

Помимо унарных, в наше распоряжение предоставлен стандартный набор бинарных операций. Операции целочисленного деления и вычисления остатка, обозначенные `quo_num` и `mod_num`, не являются инфиксными. В то же время, большинство прочих бинарных операций определены как инфиксные с именами, полученными добавлением символа “/” к именам аналогичных операций над типом `int`. Важно помнить, что в общем случае требуется использовать `=/` для

сравнения чисел. Причина этого в том, что конструкторы типа `num`, как всегда, различны по определению, но могут пересекаться по интерпретации представляемых значений. Например, в ходе вычисления частного 2^{30} и 2 при помощи двух доступных операций деления мы можем получить результаты, численно равные, но различные с точки зрения языка, поскольку в одном случае будет использован конструктор `Int`, а в другом — `Ratio`.

```
#(Int 2 **/ Int 30) // Int 2 = quo_num (Int 2 **/ Int 30) (Int 2);;
it : bool = false
#(Int 2 **/ Int 30) // Int 2 /= quo_num (Int 2 **/ Int 30) (Int 2);;
it : bool = true
```

Полный список инфиксных операций:

Оператор	Тип	Значение
<code>**/</code>	<code>num -> num -> num</code>	Вычисление степени
<code>*/</code>	<code>num -> num -> num</code>	Умножение
<code>+/</code>	<code>num -> num -> num</code>	Сложение
<code>-/</code>	<code>num -> num -> num</code>	Вычитание
<code>=/</code>	<code>num -> num -> bool</code>	Равенство
<code><>/</code>	<code>num -> num -> bool</code>	Неравенство
<code></</code>	<code>num -> num -> bool</code>	Меньше, чем
<code><=/</code>	<code>num -> num -> bool</code>	Меньше либо равно
<code>>/</code>	<code>num -> num -> bool</code>	Больше, чем
<code>>=/</code>	<code>num -> num -> bool</code>	Больше либо равно

Пример использования операций:

```
#Int 5 */ Int 14;;
it : num = Int 70
#Int 2 **/ Int 30;;
it : num = Big_int <abstr>
#(Int 2 **/ Int 30) // Int 2;;
it : num = Ratio <abstr>
#quo_num (Int 2 **/ Int 30) (Int 2);;
it : num = Int 536870912
```

Отметим, что числа типа `num` по умолчанию не выводятся. Однако, мы всегда можем преобразовать их в строки при помощи функции `string_of_num`.

```
#string_of_num(Int 2 ** Int 150);;
- : string = "1427247692705959881058285969449495136382746624"
```

Аналогично делается и обратное преобразование, реализуемое функцией с вполне естественным именем `num_of_string`.

9.3.3 Основные операции

Напомним, что для вещественных чисел нами было выбрано представление в виде функций $\mathbb{Z} \rightarrow \mathbb{Z}$. Реализация на языке ML в действительности будет использовать `int -> num`, поскольку диапазона значений встроеного целочисленного типа вполне

достаточно для задания требуемой разрядности. Определим некоторые базовые операции над вещественными числами. Наиболее фундаментальная операция, с которой мы начнём, ставит в соответствие заданному целому вещественное число. Её реализация проста:

```
#let real_of_int k n = (Int 2 **/ Int n) */ Int k;;
real_of_int : int -> int -> num = <fun>
#real_of_int 23;;
- : int -> num = <fun>
```

Очевидно, что для произвольного k справедлив критерий аппроксимации:

$$|f_k(n) - 2^n k| = |2^n k - 2^n k| = 0 < 1$$

Определим первую нетривиальную операцию — смену знака.

```
let real_neg f n = minus_num(f n);;
```

Компилятор для этой функции выводит более общий тип, чем требуется, но это не создаст трудностей. Достаточно легко убедиться, что критерий аппроксимации не нарушается. Если нам известно, что для любого n выполняется

$$|f_x(n) - 2^n x| < 1,$$

то из этого следует

$$\begin{aligned} |f_{-x}(n) - 2^n(-x)| &= | - f_x(n) - 2^n(-x) | \\ &= | - (f_x(n) - 2^n x) | \\ &= |f_x(n) - 2^n x| \\ &< 1 \end{aligned}$$

Аналогично, мы можем определить вычисление абсолютной величины вещественных чисел, используя функцию `abs_num`:

```
let real_abs f n = abs_num (f n);;
```

Доказательство корректности этого определения также не представляет трудности, принимая во внимание, что $||x| - |y|| \leq |x - y|$.

Перейдем к следующей задаче — сложению двух вещественных чисел. Предположим, что x и y представлены как f_x и f_y соответственно. Определим сложение так:

$$f_{x+y}(n) = f_x(n) + f_y(n).$$

Однако, такое определение не гарантирует соблюдения критерия аппроксимации:

$$\begin{aligned} |f_{x+y}(n) - 2^n(x + y)| &= |f_x(n) + f_y(n) - 2^n(x + y)| \\ &\leq |f_x(n) - 2^n x| + |f_y(n) - 2^n y| \end{aligned}$$

Можно утверждать, что сумма в правой части неравенства не превышает 2, в то время, как критерий ограничивает нас 1. Следовательно, в данном случае нам

требуется вычислить x и y с *большей* разрядностью, чем требуется от результата операции. Предположим, что

$$f_{x+y}(n) = (f_x(n+1) + f_y(n+1))/2.$$

Это, в свою очередь, дает

$$\begin{aligned} |f_{x+y}(n) - 2^n(x+y)| &= |(f_x(n+1) + f_y(n+1))/2 - 2^n(x+y)| \\ &\leq |f_x(n+1)/2 - 2^n x| + |f_y(n+1)/2 - 2^n y| \\ &= \frac{1}{2}|f_x(n+1) - 2^{n+1}x| + \frac{1}{2}|f_y(n+1) - 2^{n+1}y| \\ &< \frac{1}{2}1 + \frac{1}{2}1 = 1 \end{aligned}$$

Очевидно, что такое определение достигает желаемой точности. Однако, в нём неявно используется операция деления вещественных чисел. Поскольку функция должна возвращать целочисленный результат, частное требуется округлить. Если мы вычислим частное при помощи `quo_num`, ошибка округления составит почти 1 и не позволит достичь требуемой точности независимо от точности вычисления аргументов. В то же время, затратив чуть больше усилий, мы можем определить функцию деления, которая всегда будет возвращать целое число, ближайшее к точному результату (или одно из двух ближайших, если расстояние до них оказывается одинаковым), так что ошибка округления никогда не превысит $\frac{1}{2}$. Такая функция может быть реализована в целочисленной арифметике, но будет проще всего воспользоваться операцией деления рациональных чисел с последующим округлением частного к ближайшему целому, поскольку для этих действий уже определены встроенные функции:

```
#let ndiv x y = round_num(x // y);
ndiv : num -> num -> num = <fun>
##infix "ndiv";
#(Int 23) ndiv (Int 5);;
- : num = Int 5
#(Int 22) ndiv (Int 5);;
- : num = Int 4
#(Int(-11)) ndiv (Int 4);;
- : num = Int -3
#(Int(-9)) ndiv (Int 4);;
- : num = Int -2
```

Если мы определим операцию сложения с учётом сказанного выше,

$$f_{x+y}(n) = (f_x(n+2) + f_y(n+2)) \text{ ndiv } 4,$$

всё будет работать, как требуется:

$$\begin{aligned} |f_{x+y}(n) - 2^n(x+y)| &= |((f_x(n+2) + f_y(n+2)) \text{ ndiv } 4) - 2^n(x+y)| \\ &\leq \frac{1}{2} + |(f_x(n+2) + f_y(n+2))/4 - 2^n(x+y)| \\ &= \frac{1}{2} + \frac{1}{4}|(f_x(n+2) + f_y(n+2)) - 2^{n+2}(x+y)| \end{aligned}$$

$$\begin{aligned}
&\leq \frac{1}{2} + \frac{1}{4}|f_x(n+2) - 2^{n+2}x| + \frac{1}{4}|f_y(n+2) - 2^{n+2}y| \\
&< \frac{1}{2} + \frac{1}{4}1 + \frac{1}{4}1 \\
&= 1
\end{aligned}$$

Программная реализация выглядит так:

```
let real_add f g n =
  (f(n + 2) +/ g(n + 2)) ndiv (Int 4);;
```

Аналогичные рассуждения могут быть использованы для определения операции вычитания, но проще всего построить её на основе уже введенных функций:

```
#let real_sub f g = real_add f (real_neg g);;
real_sub : (num -> num) -> (num -> num) -> num -> num = <fun>
```

Реализация умножения, вычисления обратных чисел и деления потребует в общем случае несколько больших усилий. Однако, частные случаи умножения и деления на целое число существенно легче и при этом достаточно часты. По соображениям эффективности они заслуживают отдельного рассмотрения. Пусть

$$f_{mx}(n) = (mf_x(n + p + 1)) \text{ ndiv } 2^{p+1},$$

где p выбирается так, чтобы $2^p \geq |m|$. Доказать корректность этого определения легко:

$$\begin{aligned}
|f_{mx}(n) - 2^n(mx)| &\leq \frac{1}{2} + \left| \frac{mf_x(n + p + 1)}{2^{p+1}} - 2^n(mx) \right| \\
&= \frac{1}{2} + \frac{|m|}{2^{p+1}} |f_x(n + p + 1) - 2^{n+p+1}x| \\
&< \frac{1}{2} + \frac{|m|}{2^{p+1}} \\
&\leq \frac{1}{2} + \frac{1}{2} \frac{|m|}{2^p} \\
&\leq \frac{1}{2} + \frac{1}{2} = 1.
\end{aligned}$$

Для реализации такого подхода нам потребуется функция вычисления соответствующего p . Не слишком изощрённое, но вполне подходящее определение может выглядеть так:

```
let log2 =
  let rec log2 x y =
    if x </ Int 1 then y
    else log2 (quo_num x (Int 2)) (y + 1) in
  fun x -> log2 (x -/ Int 1) 0;;
```

С учётом сказанного выше, операция умножения на целое число принимает вид:

```

let real_intmul m x n =
  let p = log2 (abs_num m) in
  let p1 = p + 1 in
  (m */ x(n + p1)) ndiv (Int 2 **/ Int p1);;

```

Деление на целое число вводится следующим образом:

$$f_{x/m}(n) = f_x(n) \text{ ndiv } m$$

Для упрощения доказательства корректности этого определения будем считать, что случай $m = 0$ никогда не может возникнуть, а при $m = \pm 1$ результат операции не изменяет погрешности. В остальных случаях из $|f_x(n) - 2^n x| < 1$ следует, что $|f_x(n)/m - 2^n x/m| < \frac{1}{|m|} \leq \frac{1}{2}$, откуда, в свою очередь, с учётом $|f_x(n) \text{ ndiv } m - f_x(n)/m| \leq \frac{1}{2}$ получаем требуемое. В итоге, программная реализация деления такова:

```

let real_intdiv m x n =
  x(n) ndiv (Int m);;

```

9.3.4 Умножение: общий случай

Определить умножение в общем случае труднее, поскольку погрешность аппроксимации одного из сомножителей умножается на порядок второго. Следовательно, нам потребуется предварительно оценить порядки сомножителей. Поступим следующим образом. Предположим, что нам требуется вычислить выражение $x + y$ до n -го разряда. Для этого выберем r и s такие, что $|r - s| \leq 1$ и $r + s = n + 2$. Таким образом, обе эти величины несколько больше, чем половина требуемой разрядности. Далее вычислим $f_x(r)$ и $f_y(s)$, после чего определим p и q — соответствующие «двоичные логарифмы», для которых справедливо $|f_x(r)| \leq 2^p$ and $|f_y(s)| \leq 2^q$. Если как p , так и q равны нулю, то легко убедиться, что результатом вычислений также будет 0. В противном случае отметим, что либо $p > 0$, либо $q > 0$ — этот факт нам понадобится в дальнейшем.

Пусть

$$\begin{aligned}
 k &= n + q - s + 3 = q + r + 1 \\
 l &= n + p - r + 3 = p + s + 1 \\
 m &= (k + l) - n = p + q + 4.
 \end{aligned}$$

Покажем, что погрешность выражения $f_{xy}(n) = (f_x(k)f_y(l)) \text{ ndiv } 2^m$ удовлетворяет критерию аппроксимации, т. е. $|f_{xy}(n) - 2^n(xy)| < 1$. Если ввести обозначения

$$\begin{aligned}
 2^k x &= f_x(k) + \delta, \\
 2^l y &= f_y(l) + \epsilon,
 \end{aligned}$$

где $|\delta| < 1$ и $|\epsilon| < 1$, то мы получим:

$$|f_{xy}(n) - 2^n(xy)| \leq \frac{1}{2} + \left| \frac{f_x(k)f_y(l)}{2^m} - 2^n(xy) \right|$$

$$\begin{aligned}
&= \frac{1}{2} + 2^{-m} |f_x(k)f_y(l) - 2^{k+l}xy| \\
&= \frac{1}{2} + 2^{-m} |f_x(k)f_y(l) - (f_x(k) + \delta)(f_y(l) + \epsilon)| \\
&= \frac{1}{2} + 2^{-m} |\delta f_y(l) + \epsilon f_x(k) + \delta\epsilon| \\
&\leq \frac{1}{2} + 2^{-m} (|\delta f_y(l)| + |\epsilon f_x(k)| + |\delta\epsilon|) \\
&\leq \frac{1}{2} + 2^{-m} (|f_y(l)| + |f_x(k)| + |\delta\epsilon|) \\
&< \frac{1}{2} + 2^{-m} (|f_y(l)| + |f_x(k)| + 1)
\end{aligned}$$

Отсюда имеем $|f_x(r)| \leq 2^p$, так что $|2^r x| < 2^p + 1$. Следовательно, $|2^k x| < 2^{q+1}(2^p + 1)$, откуда $|f_x(k)| < 2^{q+1}(2^p + 1) + 1$, т. е. $|f_x(k)| \leq 2^{q+1}(2^p + 1)$. Аналогично доказывается справедливость $|f_y(l)| \leq 2^{p+1}(2^q + 1)$. Таким образом,

$$\begin{aligned}
|f_y(l)| + |f_x(k)| + 1 &\leq 2^{p+1}(2^q + 1) + 2^{q+1}(2^p + 1) + 1 \\
&= 2^{p+q+1} + 2^{p+1} + 2^{p+q+1} + 2^{q+1} + 1 \\
&= 2^{p+q+2} + 2^{p+1} + 2^{q+1} + 1.
\end{aligned}$$

Для получения требуемой максимальной погрешности введём ограничение $|f_y(l)| + |f_x(k)| + 1 \leq 2^{m-1}$, либо, разделив на 2 и учитывая дискретность множества целых чисел,

$$2^{p+q+1} + 2^p + 2^q < 2^{p+q+2}.$$

В свою очередь, данное отношение может быть записано как $(2^{p+q} + 2^p) + (2^{p+q} + 2^q) < 2^{p+q+1} + 2^{p+q+1}$. Его справедливость следует из упомянутого ранее факта, что либо $p > 0$, либо $q > 0$. Таким образом, мы обосновали следующее определение:

```

let real_mul x y n =
  let n2 = n + 2 in
  let r = n2 / 2 in
  let s = n2 - r in
  let xr = x(r)
  and ys = y(s) in
  let p = log2 xr
  and q = log2 ys in
  if p = 0 & q = 0 then Int 0 else
  let k = q + r + 1
  and l = p + s + 1
  and m = p + q + 4 in
  (x(k) */ y(l)) ndiv (Int 2 **/ Int m);;

```

9.3.5 Обратные числа

Нашим следующим шагом будет реализация вычисления обратных чисел. Чтобы получить любую верхнюю оценку обратного числа, не говоря уж о хорошем его приближении, потребуется оценить аргумент *снизу*. В общем случае для этого не

существует лучшего способа, чем вычисление аргумента с возрастающей точностью, пока we can bound it away from zero. Следующая лемма служит обоснованием этой процедуры.

Лемма 9.1 Пусть $2e \geq n + k + 1$, $|f_x(k)| \geq 2^e$ и $|f_x(k) - 2^k x| < 1$, где $f_x(k)$ — целое, а e , n и k — натуральные числа. Если мы определим

$$f_y(n) = 2^{n+k} \text{ ndiv } f_x(k),$$

то получим $|f_y(n) - 2^n x^{-1}| < 1$, т. е. требуемую верхнюю оценку погрешности.

Доказательство: Доказательство этой леммы достаточно утомительно и будет приведено здесь не полностью, а лишь в виде основных соображений. Если $|f_x(k)| > 2^e$, то результат очевидно следует из простых рассуждений. Округление в результате выполнения операции `ndiv` даёт погрешность, не превышающую $\frac{1}{2}$, итоговая погрешность меньше $\frac{1}{2}$. Если $|f_x(k)| = 2^e$, но при этом $n + k \geq e$, то можно пренебречь тем, что второй компонент погрешности может быть вдвое большим, т. е. меньшим 1 — в этом случае ошибка округления будет отсутствовать потому, что $f_x(k) = \pm 2^e$ делится на 2^{n+k} нацело. (Воспользуемся фактом, что $2^e - 1 \leq 2^{e-1}$, поскольку при $2e \geq n + k + 1$ значение e не может равняться нулю.) Наконец, при $|f_x(k)| = 2^e$ и $n + k < e$, мы имеем $|f_y(n) - 2^n \frac{1}{x}| < 1$, поскольку $|f_y(n)| \leq 1$ и $0 < |2^n \frac{1}{x}| < 1$, а знаки этих величин совпадают. \square

Предположим, что нам требуется найти число, обратное x , с точностью n . Прежде всего, вычислим $f_x(0)$. Нам потребуется рассмотреть два случая:

1. Если $|f_x(0)| > 2^r$ для некоторого натурального числа r , то выберем наименьшее натуральное число k (которое может быть и равным нулю) такое, что $2r + k \geq n + 1$, и положим $e = r + k$. Результатом вычислений в этом случае будет $2^{n+k} \text{ ndiv } f_x(k)$. Легко убедиться, что условия, требуемые леммой, выполняются. Поскольку $|f_x(0)| \geq 2^r + 1$, мы имеем $|x| > 2^r$, так что $|2^k x| > 2^{r+k}$. Это значит, что $|f_x(k)| > 2^{r+k} - 1$, а отсюда (учитывая целочисленность $f_x(k)$) получаем требуемое соотношение $|f_x(k)| \geq 2^{r+k} = 2^e$. Условие $2e \geq n = k + 1$ легко проверить. Отметим, что из $r \geq n$ непосредственно следует корректность аппроксимации $f_y(n) = 0$.
2. При $|f_x(0)| \leq 1$ воспользуемся функцией `'msd'`, возвращающей наименьшее p такое, что $|f_x(p)| > 1$. Отметим, что при $x = 0$ произойдёт заикливание. Положив $e = n + p + 1$ и $k = e + p$, определим результат операции как $2^{n+k} \text{ ndiv } f_x(k)$. В этом случае также справедливы условия, требуемые леммой. Так как $|f_x(p)| \geq 2$, мы имеем $|2^p x| > 1$, т. е. $|x| > \frac{1}{2^p}$. Следовательно $|2^k x| > 2^{k-p} = 2^e$, откуда $|f_x(k)| > 2^e - 1$, т. е. $|f_x(k)| \geq 2^e$.

Реализацию начнём с функции `msd`:

```
let msd =
  let rec msd n x =
    if abs_num(x(n)) > Int 1 then n else msd (n + 1) x in
  msd 0;;
```

после чего воплотим изложенные выше теоретические рассуждения в виде простой программы:

```

let real_inv x n =
  let x0 = x(0) in
  let k =
    if x0 >/ Int 1 then
      let r = log2 x0 - 1 in
      let k0 = n + 1 - 2 * r in
      if k0 < 0 then 0 else k0
    else
      let p = msd x in
      n + 2 * p + 1 in
  (Int 2 **/ Int (n + k)) ndiv (x(k));;

```

В итоге определение операции деления становится тривиальным:

```

let real_div x y = real_mul x (real_inv y);;

```

9.3.6 Отношения порядка

Примечательным свойством отношений порядка является то, что их вычисление в общем случае алгоритмически неразрешимо. В основе такого вывода лежит невозможность установить в общем случае равенство данного числа нулю. Если последовательное вычисление с увеличением требуемой точности продолжает выдавать 0, это ещё не служит гарантией того, что в дальнейшем мы не получим ненулевой результат.⁵ Если значение x не равно нулю, поиск первого ненулевого разряда числа когда-либо завершится, но в случае $x = 0$ он будет длиться вечно.

Принимая во внимание сказанное выше, несложно реализовать отношения порядка. Для определения взаимного порядка чисел x и y достаточно найти n такое, что $|x_n - y_n| \geq 2$. Например, для $x_n \geq y_n + 2$ мы имеем

$$2^n x > x_n - 1 \geq y_n + 1 > 2^n y.$$

откуда делаем вывод, что $x > y$. Прежде всего, приведем общую процедуру вычисления n , после чего все отношения порядка могут быть выражены с её использованием. Отметим, что единственным способом реализации рефлексивных отношений будет положить их тождественными соответствующим нерелексивным отношениям!

⁵Доказательство алгоритмической неразрешимости этой задачи сводится к проблеме завершимости: пусть $f(n) = 1$, если некоторая машина Тьюринга завершает работу не более чем за n итераций, и $f(n) = 0$ в противном случае.

```

let separate =
  let rec separate n x y =
    let d = x(n) -/ y(n) in
    if abs_num(d) >/ Int 1 then d
    else separate (n + 1) x y in
  separate 0;;

let real_gt x y = separate x y >/ Int 0;;
let real_ge x y = real_gt x y;;
let real_lt x y = separate x y </ Int 0;;
let real_le x y = real_lt x y;;

```

9.3.7 Кэширование

Чтобы протестировать определённые нами функции, потребуется возможность вывода некоторого приближённого значения вещественного числа в десятичной системе счисления. Возможности стандартной библиотеки CAML делают эту задачу несложной. Если нам требуется вывести d десятичных знаков числа, положим точность вычислений n такой, чтобы $2^n > 10^d$, т. е. точность вычислений была бы не меньшей, чем количество выводимых цифр.

```

let view x d =
  let n = 4 * d in
  let out = x(n) // (Int 2 **/ Int n) in
  approx_num_fix d out;;

```

Начнём с простых примеров, которые работают, как ожидается:

```

#let x = real_of_int 3;;
x : int -> num = <fun>
#let xi = real_inv x;;
xi : int -> num = <fun>
#let wun = real_mul x xi;;
wun : int -> num = <fun>
#view x 20;;
it : string = "3.00000000000000000000"
#view xi 20;;
it : string = ".33333333333333333333"
#view wun 20;;
it : string = "1.00000000000000000000"

```

Однако, дальнейшее тестирование обнаруживает в нашей реализации серьёзную трудноуловимую проблему, которая проявляется с ростом сложности задач. Эта проблема — многократное вычисление одних и тех же значений. Помимо очевидного случая — явного наличия общих подвыражений, многократное вычисление одних и тех же выражений с различной точностью требуется в алгоритмах умножения и, в особенности, вычисления обратного числа. Количество обращений к одному и тому же подвыражению зависит от глубины его вложенности в исходное выражение и может расти экспоненциально:

```
#let x1 = real_of_int 1 in
  let x2 = real_mul x1 x1 in
    let x3 = real_mul x2 x2 in
      let x4 = real_mul x3 x3 in
        let x5 = real_mul x4 x4 in
          let x6 = real_mul x5 x5 in
            let x7 = real_mul x6 x6 in
              view x7 10;;
            - : string = "+1.0000000000"
```

Вычисление этого примера может занять несколько секунд.

Для решения проблемы воспользуемся идеей *кэширования* или *функций с памятью* (Michie 1968). Каждой функции поставим в соответствие ячейку памяти, (???ref. cell) в которой будем хранить её значение, вычисленное с наибольшей на данный момент точностью. При очередном обращении к функции с той же самой требуемой точностью это значение может быть возвращено немедленно, без каких-либо повторных вычислений. Кроме того, менее точная аппроксимация (например, порядка n) всегда может быть получена из более точной ($n + k$, где $k \geq 1$). Если нам известно, что $|f_x(n + k) - 2^{n+k}x| < 1$, из этого следует:

$$\begin{aligned}
 |f_x(n + k) \text{ ndiv } 2^k - 2^n x| &\leq \frac{1}{2} + \left| \frac{f_x(n + k)}{2^k} - 2^n x \right| \\
 &= \frac{1}{2} + \frac{1}{2^k} |f_x(n + k) - 2^{n+k} x| \\
 &< \frac{1}{2} + \frac{1}{2^k} \\
 &\leq 1.
 \end{aligned}$$

Таким образом, использование $f_x(n + k) \text{ ndiv } 2^k$ в качестве аппроксимации порядка n обосновано.

Для реализации функций с памятью воспользуемся обобщённой функцией `memo`, которую требуется добавить во все определённые ранее операции над вещественными числами:


```

let real_of_int k = memo (fun n -> (Int 2 **/ Int n) */ Int k);;

let real_neg f = memo (fun n -> minus_num(f n));;

let real_abs f = memo (fun n -> abs_num (f n));;

let real_add f g = memo (fun n ->
  (f(n + 2) +/ g(n + 2)) ndiv (Int 4));;

let real_sub f g = real_add f (real_neg g);;

let real_intmul m x = memo (fun n ->
  let p = log2 (abs_num m) in
  let p1 = p + 1 in
  (m */ x(n + p1)) ndiv (Int 2 **/ Int p1));;

let real_intdiv m x = memo (fun n ->
  x(n) ndiv (Int m));;

let real_mul x y = memo (fun n ->
  let n2 = n + 2 in
  let r = n2 / 2 in
  let s = n2 - r in
  let xr = x(r)
  and ys = y(s) in
  let p = log2 xr
  and q = log2 ys in
  if p = 0 & q = 0 then Int 0 else
  let k = q + r + 1
  and l = p + s + 1
  and m = p + q + 4 in
  (x(k) */ y(l)) ndiv (Int 2 **/ Int m));;

let real_inv x = memo (fun n ->
  let x0 = x(0) in
  let k =
    if x0 >/ Int 1 then
      let r = log2 x0 - 1 in
      let k0 = n + 1 - 2 * r in
      if k0 < 0 then 0 else k0
    else
      let p = msd x in
      n + 2 * p + 1 in
  (Int 2 **/ Int (n + k)) ndiv (x(k)));;

let real_div x y = real_mul x (real_inv y);;

```

где

```

let memo f =
  let mem = ref (-1, Int 0) in
  fun n -> let (m, res) = !mem in
            if n <= m then
              if m = n then res
              else res ndiv (Int 2 **/ Int(m - n))
            else
              let res = f n in
              mem := (n, res); res;;

```

Проведенная оптимизация делает вычисление упомянутого выше произведения практически мгновенным. Рассмотрим еще несколько примеров:

```

#let pi1 = real_div (real_of_int 22) (real_of_int 7);;
pi1 : int -> num = <fun>
#view pi1 10;;
it : string = "+3.1428571429"
#let pi2 = real_div (real_of_int 355) (real_of_int 113);;
pi2 : int -> num = <fun>
#view pi2 10;;
it : string = "+3.1415929204"
#let pidiff = real_sub pi1 pi2;;
pidiff : int -> num = <fun>
#view pidiff 20;;
it : string = "+0.00126422250316055626"
#let ipidiff = real_inv pidiff;;
ipidiff : int -> num = <fun>
#view ipidiff 20;;
it : string = "+791.00000000000000000000"

```

В заключение отметим, что все расчеты, приведенные в данном разделе, можно, безусловно, проделать и в рациональной арифметике. Но на деле может оказаться, что наш подход более эффективен в некоторых ситуациях, так как он избавляет от свойственного вычислениям в рациональных числах лавинообразного роста числителей и знаменателей, который абсолютно избыточен, когда нам нужно лишь приближенное значение результата. Однако, предложенный метод раскрывает в полной мере свои достоинства лишь тогда, когда нам потребуется ввести трансцендентные функции наподобие \exp , \sin и т. д. Этот вопрос здесь рассматриваться не будет ввиду ограничений на объем курса, но может оказаться интересным в качестве упражнения. Одним из подходов является применение частичных сумм соответствующих рядов Тейлора. Отметим, что конечные суммы могут быть вычислены напрямую вместо итеративного применения функции сложения — это существенно улучшает их точность.

9.4 Пролог и доказательство теорем

Язык Пролог популярен в области исследований по Искусственному Интеллекту, и используется в различных практических приложениях, таких как экспертные системы и интеллектуальные базы данных. В этой главе рассматривается как различные механизмы Пролога, а именно: поиск вглубь с унификацией и возвратом,

могут быть реализованы в ML. Мы не претендуем на полную реализацию Пролога, но результатов этого раздела достаточно, чтобы получить точное представление об основных достоинствах языка и запустить несколько примеров.

9.4.1 Термы пролога

Данные и код в Прологе представляются с помощью единой системы термов первого порядка. Мы уже определяли тип термов для математических выражений, их печати, и соответствующих синтаксических анализаторов. Сейчас мы будем использовать нечто похожее, но с некоторыми модификациями. Во-первых, немного упростим код, будем рассматривать константы как нуль-арные функции, т.е. как функции, принимающие пустой список аргументов. Соответственно определим:

```
type term = Var of string
          | Fn of string * (term list) ;;
```

Там, где раньше использовался `Const s`, теперь будет `Fn(s, [])`. Обратите внимание, функции различной арности (разным числом аргументов) рассматриваются как различные, даже если у них одинаковое имя. Следовательно нет опасности, что константы будут спутаны с настоящими функциями.

9.4.2 Лексический анализ

В целях следования соглашениям языка, которые включают чувствительность к регистру, мы обновим функции лексического анализа. Не будем требовать точного совпадения во всём, но учтём самое главное: символьно-цифровые идентификаторы, начинающиеся с буквы в *верхнем регистре* или знака подчёркивания, рассматриваются как переменные, а другие символьно-цифровые идентификаторы и числа – как константы. Например, `X` и `Answer` – переменные, тогда как `x` и `john` – константы. Символьные идентификаторы, также как константы не различаются, в отличие от символов пунктуации: левой и правой скобок, запятой и точки с запятой – их различать будем. Непунктуационные символы собираются вместе до строки наибольшей возможной длины.

```
type token = Variable of string
          | Constant of string
          | Punct of string ;;
```

Лексический анализатор, следовательно, выглядит так:

```

let lex =
  let several p = many (some p) in
  let collect(h,t) = h^(itlist (prefix ^) t "") in
  let upper_alpha s = "A" <= s & s <= "Z" or s = "_"
  and lower_alpha s = "a" <= s & s <= "z" or "0" <= s & s <= "9"
  and punct s = s = "(" or s = ")" or s = "[" or s = "]"
                  or s = "," or s = "."
  and space s = s = " " or s = "\n" or s = "\t" in
  let alphanumeric s = upper_alpha s or lower_alpha s in
  let symbolic s = not space s & not alphanumeric s & not punct s in
  let rawvariable =
    some upper_alpha ++ several alphanumeric >> (Variable o collect)
  and rawconstant =
    (some lower_alpha ++ several alphanumeric ||
     some symbolic ++ several symbolic) >> (Constant o collect)
  and rawpunct = some punct >> Punct in
  let token =
    (rawvariable || rawconstant || rawpunct) ++
    several space >> fst in
  let tokens = (several space ++ many token) >> snd in
  let alltokens = (tokens ++ finished) >> fst in
  fst o alltokens o explode;;

```

Например:

```

#lex "add(X,Y,Z) :- X is Y+Z.";;
- : token list =
[Constant "add"; Punct "("; Variable "X"; Punct ",";
 Variable "Y"; Punct ","; Variable "Z"; Punct ")";
 Constant ":-"; Variable "X"; Constant "is"; Variable "Y";
 Constant "+"; Variable "Z"; Punct "."]

```

9.4.3 Синтаксический анализ

Основной синтаксический анализатор остаётся в значительной степени таким же, каким был ранее, система печати не меняется. Единственная модификация – прологовские списки записываются в более удобной нотации. В Прологе «.» и «nil» соответствуют «::» и «[]» в ML. Мы также допускаем прологовскую запись «[H|T]» вместо «cons(H,T)», используемую для сравнения по образцу. После основных функций:

```

let variable =
  fun (Variable s::rest) -> s,rest
  | _ -> raise Noparse;;

let constant =
  fun (Constant s::rest) -> s,rest
  | _ -> raise Noparse;;

```

у нас есть синтаксический анализатор для термов, а также правил Пролога, имеющих следующий вид:

$$\begin{aligned} &term. \\ &term \quad :- \quad term_1, \dots, term_n. \end{aligned}$$

Собственно анализаторы:

```

let rec atom input
  = (constant ++ a (Punct "(" ++ termlist ++ a (Punct ")")
    >> (fun ((name, _), args), _) -> Fn(name, args))
  || constant
    >> (fun s -> Fn(s, []))
  || variable
    >> (fun s -> Var s)
  || a (Punct "(" ++ term ++ a (Punct ")")
    >> (snd o fst)
  || a (Punct "[" ++ list
    >> snd) input
and term input = precedence (!infixes) atom input
and termlist input
  = (term ++ a (Punct ",") ++ termlist
    >> (fun ((h, _), t) -> h :: t)
  || term
    >> (fun h -> [h])) input
and list input
  = (term ++ (a (Constant "]" ++ term ++ a (Punct "[")
    >> (snd o fst)
  || a (Punct ",") ++ list
    >> snd
  || a (Punct "]" ++
    >> (K (Fn(" [] ", []))))
    >> (fun (h, t) -> Fn(".", [h; t]))
  || a (Punct "]" ++
    >> (K (Fn(" [] ", [])))) input
and rule input
  = (term ++ (a (Punct ".")
    >> (K []))
  || a (Constant ":-") ++ term ++
    many (a (Punct ",") ++ term >> snd) ++
    a (Punct ".")
    >> (fun (((_, h), t), _) -> h :: t))) input;;

let parse_term = fst o (term ++ finished >> fst) o lex;;

let parse_rules = fst o (many rule ++ finished >> fst) o lex;;

```

9.4.4 Унификация

Пролог использует набор правил для разрешения текущей *цели*. Для этого он подбирает правила для цели. Правило, состоящее из одного терма, разрешает цель немедленно, если оно совпадает с целью. В случае правила вида $term :- term_1, \dots, term_n$, если цель совпадает с $term$, то требуется разрешить, как подцель, по порядку каждый $term_i$. Цель является разрешимой, когда это выполнимо.

Однако, цели и правила не обязаны совпадать в точности до терма. В состав цели и сопоставляемого правила могут входить переменные, и при сравнении цели с правилом переменные могут присваивать (правильно было бы говорить связывать) соответствующие значения. Такое сравнение со связыванием называется *унификацией*. Оно означает, что мы можем пренебречь проверкой отдельных вариантов исходной цели, т.е. ограничиться $P(Y)$ вместо $P(f(X))$. Например:

- Для унификации $f(g(X), Y)$ и $f(g(a), X)$ связываются $X = a$ и $Y = a$. После этого термы будут: $f(g(a), a)$.
- Для унификации $f(a, X, Y)$ и $f(X, a, Z)$ связываются $X = a$ и $Y = Z$, после чего оба терма примут вид $f(a, a, Z)$.
- Недопустимо унифицировать $f(X)$ и X ().

В общем случае, унификация неоднозначна. Например, во втором примере можно выбрать связывание $Y = f(b)$ и $Z = f(b)$. Однако, всегда предпочтительнее *наиболее общая* унификация, из которой остальные выводятся дальнейшей конкретизацией (сравните с наиболее общими типами в ML). Для того чтобы найти её, требуется рекурсивно (и синхронно) обходить вглубь оба терма, и при нахождении переменной в одном терме, связывать её с соответствующим подтермом другого терма. При этом также требуется следить, чтобы уже связанные переменные не унифицировались повторно с отличными термами и чтобы не возникало ситуаций, как в последнем примере, когда переменная, связываемая с термом входит в него же. Простая реализация этой идеи такова:

```

let rec unify tm1 tm2 insts =
  match tm1 with
    Var(x) ->
      (try let tm1' = assoc x insts in
        unify tm1' tm2 insts
       with Not_found ->
        augment (x,tm2) insts)
  | Fn(f1, args1) ->
    match tm2 with
      Var(y) ->
        (try let tm2' = assoc y insts in
          unify tm1 tm2' insts
         with Not_found ->
          augment (y,tm1) insts)
    | Fn(f2, args2) ->
      if f1 = f2
      then itlist2 unify args1 args2 insts
      else raise (error "functions_do_not_match");;

```

Аккумулятор переменных `insts` должен пополняться с «проверкой наличия». http://en.wikipedia.org/wiki/Occurs_check А именно, мы должны запретить связывания переменной X с нетривиальным термом, включающим X , как в третьем примере выше. Большинство реализаций Пролога игнорируют подобную проверку или по (заявленным) причинам эффективности, или для поддержки циклических структур данных, более сложных чем простые термы первого порядка.

```

let rec occurs_in x =
  fun (Var y) -> x = y
    | (Fn(_, args)) -> exists (occurs_in x) args;;

let rec subst insts = fun
  (Var y) -> (try assoc y insts with Not_found -> tm)
  | (Fn(f, args)) -> Fn(f, map (subst insts) args);;

let raw_augment =
  let augment1 theta (x,s) =
    let s' = subst theta s in
    if occurs_in x s & not(s = Var(x))
    then raise (error "Occurs_check")
    else (x,s') in
  fun p insts -> p::(map (augment1 [p]) insts);;

let augment (v,t) insts =
  let t' = subst insts t in match t' with
  Var(w) -> if w <= v then
    if w = v then insts
    else raw_augment (v,t') insts
    else raw_augment (w,Var(v)) insts
  | _ -> if occurs_in v t'
    then raise (error "Occurs_check")
    else raw_augment (v,t') insts;;

```

9.4.5 Перебор с возвратами

Пролог выполняет поиск в глубину, но он может использовать возврат: допустим правило успешно унифицирует цель, но последующие цели не могут быть разрешены при текущих связываниях. Тогда выполняется возврат, связывание переменных освобождается, и с текущей целью сверяется другое правило.


```

let rec first f =
  fun [] -> raise (error "No_rules_applicable")
  | (h::t) -> try f h with error _ -> first f t;;

let rec expand n rules insts goals =
  first (fun rule ->
    if goals = [] then insts else
    let conc,asms =
      rename_rule (string_of_int n) rule in
    let insts' = unify conc (hd goals) insts in
    let local,global = partition
      (fun (v,_) -> occurs_in v conc or
        exists (occurs_in v) asms) insts' in
    let goals' = (map (subst local) asms) @
      (tl goals) in
    expand (n + 1) rules global goals') rules;;

```

Для генерации новых переменных используется функция `rename` :

```

let rec rename s =
  fun (Var v) -> Var("~"~v~s)
  | (Fn(f,args)) -> Fn(f,map (rename s) args);;

let rename_rule s (conc,asms) =
  (rename s conc,map (rename s) asms);;

```

Наконец, соберём всё вместе, в одну функцию `prolog`, которая пытается разрешить цель, используя заданные правила:

```

type outcome = No | Yes of (string * term) list;;

let prolog rules goal =
  try let insts = expand 0 rules [] [goal] in
    Yes(filter (fun (v,_) -> occurs_in v goal)
      insts)
  with error _ -> No;;

```

Результат функции - сообщение о том, что цель или не может быть разрешена, или разрешается с заданными связываниями. В последнем случае мы возвращаем только один ответ (один набор связанных переменных), но можно изменить код так, чтобы выводились все возможные решения.

9.4.6 Примеры

Давайте испытаем написанный интерпретатор на примерах из какой-нибудь книги по Прологу. Например:

```
#let rules = parse_rules
"male(albert).
male(edward).
female(alice).
female(victoria).
parents(edward,victoria,albert).
parents(alice,victoria,albert).
sister_of(X,Y) :-
    female(X),
    parents(X,M,F),
    parents(Y,M,F).";;
rules : (term * term list) list =
['male(albert)', [], 'male(edward)', [],
'female(alice)', [], 'female(victoria)', [],
'parents(edward,victoria,albert)', [],
'parents(alice,victoria,albert)', [],
'sister_of(X,Y)',
 ['female(X)', 'parents(X,M,F)', 'parents(Y,M,F)']]
#prolog rules ("sister_of(alice,edward)");;
- : outcome = Yes []
#prolog rules (parse_term "sister_of(alice,X)");;
- : outcome = Yes ["X", 'edward']
#prolog rules (parse_term "sister_of(X,Y)");;
- : outcome = Yes ["Y", 'edward'; "X", 'alice']
```

Поскольку Пролог является скорее реляционным чем функциональным языком, можно использовать запросы в более гибкой форме, например спрашивать при каких аргументах можно получить заданный результат:

```
#let r = parse_rules
"append([],L,L).
append([H|T],L,[H|A]) :- append(T,L,A).";;
r : (term * term list) list =
['append([],L,L)', [],
'append(H . T,L,H . A)', ['append(T,L,A)']]
#prolog r (parse_term "append([1,2],[3],[1,2,3])");;
- : outcome = Yes []
#prolog r (parse_term "append([1,2],[3,4],X)");;
- : outcome = Yes ["X", '1 . (2 . (3 . (4 . [])))']
#prolog r (parse_term "append([3,4],X,X)");;
- : outcome = No
#prolog r (parse_term "append([1,2],X,Y)");;
- : outcome = Yes ["Y", '1 . (2 . X)']
```

В таких случаях Пролог представляется в значительной степени интеллектуальным. Но за этой наружностью стоит простой поиск, а значит «интеллектуальность» хрупка. Например, следующий код заикливается:

```
#prolog r (parse_term "append(X,[3,4],X)");;
```

9.4.7 Доказательство теорем

Пролог действует как простой доказыватель теорем, используя базу логических фактов (правил) для доказательства цели. Однако он весьма ограничен в

доказательных возможностях, частично из-за неполноты стратегии поиска в глубину, частично потому что он может выполнять логические выводы только определённого вида. Вообще, можно реализовать более мощную пролого-подобную систему: [Stickel \(1988\)](#). Далее мы покажем, как в сущности на тех же идеях (унификация и возврат) строится более функциональный доказыватель теорем.

Унификация является эффективным способом ограничения переменных с навешенным квантором общности. Например, имея правила $\forall X. p(X) \Rightarrow q(X)$ и $p(f(a))$, можно унифицировать оба выражения, включающие p и тем самым связать X с $f(a)$. Напротив, самые ранние системы доказательства теорем пытались строить всевозможные термы из доступных констант и функций («Эрбрановского базиса»).

Обычно поиск вглубь может привести к бесконечному заикливанию, так что мы должны незначительно изменить стратегию поиска. Будем использовать *поиск вглубь с итеративным погружением*. Это значит, что поиск вглубь имеет фиксированный предел, по достижению которого выполняется возврат. Если доказательство не найдено при заданной глубине, её значение увеличивается и делается ещё один поиск. Таким образом, сначала ищутся доказательства глубины 1, в случае провала - глубины 2, 3 и так далее. В качестве предельной глубины могут использоваться различные параметры, например, высота или размер дерева поиска; мы будем использовать число унифицируемых переменных.

Манипулирование формулами

Для обозначения формул будем использовать термы первого порядка. Добавим также новые константы для логических операторов, некоторые из которых - инфиксные.

Оператор	Значение
$\sim(p)$	не p
$p \ \& \ q$	p и q
$p \mid q$	p или q
$p \ --> \ q$	p влечёт q (импликация)
$p \ <-> \ q$	p только и если только q (эквивалентность)
$\text{forall}(X,p)$	для всех X , p (квантор общности)
$\text{exists}(X,p)$	существует X такой что p (квантор существования)

Альтернативный подход заключается в добавлении отдельного типа для формул, но тогда потребуется добавить код и для синтаксического разбора, и печати. Из соображений простоты ограничимся первым решением.

Предварительная обработка формул

Очевидно, программа упростится, если основная часть доказывателя не будет работать с импликациями и эквивалентностями. Следовательно, сначала определим функцию, заменяющую их комбинацией других операторов.

```

let rec proc tm =
  match tm with
    Fn("~", [t]) -> Fn("~", [proc t])
  | Fn("&", [t1; t2]) -> Fn("&", [proc t1; proc t2])
  | Fn("|", [t1; t2]) -> Fn("|", [proc t1; proc t2])
  | Fn("—>", [t1; t2]) ->
    proc (Fn("|", [Fn("~", [t1]); t2]))
  | Fn("<->", [t1; t2]) ->
    proc (Fn("&", [Fn("—>", [t1; t2]);
                  Fn("—>", [t2; t1])]))
  | Fn("forall", [x; t]) -> Fn("forall", [x; proc t])
  | Fn("exists", [x; t]) -> Fn("exists", [x; proc t])
  | t -> t;;

```

Определим две взаимно рекурсивные функции, которые по формуле и её отрицанию строят «негативно нормальную форму» - формулу из элементарных термов и их отрицаний, связанных с помощью операторов «И», «ИЛИ» и кванторов.

```

let rec nnf_p tm =
  match tm with
    Fn("~", [t]) -> nnf_n t
  | Fn("&", [t1; t2]) -> Fn("&", [nnf_p t1; nnf_p t2])
  | Fn("|", [t1; t2]) -> Fn("|", [nnf_p t1; nnf_p t2])
  | Fn("forall", [x; t]) -> Fn("forall", [x; nnf_p t])
  | Fn("exists", [x; t]) -> Fn("exists", [x; nnf_p t])
  | t -> t

and nnf_n tm =
  match tm with
    Fn("~", [t]) -> nnf_p t
  | Fn("&", [t1; t2]) -> Fn("|", [nnf_n t1; nnf_n t2])
  | Fn("|", [t1; t2]) -> Fn("&", [nnf_n t1; nnf_n t2])
  | Fn("forall", [x; t]) -> Fn("exists", [x; nnf_n t])
  | Fn("exists", [x; t]) -> Fn("forall", [x; nnf_n t])
  | t -> Fn("~", [t]);;

```

Идея такова: отрицание доказываемой формулы конвертируется в негативно нормальную формулу, с которой главный доказыватель попытается получить противоречие. Этого противоречия будет достаточно для доказательства исходной формулы.

Главный доказыватель

На каждом этапе доказыватель имеет текущую формулу, список формул, которые ещё предстоит рассмотреть и список литералов. Он пытается получить противоречие. Используется следующая стратегия:

- Если текущая формула – $p \ \& \ q$, то p и q рассматриваются по отдельности, т.е p делается текущей формулой, а q добавляется к формулам «на рассмотрение».

- Если текущая формула имеет вид $p \mid q$, то делается попытка получить противоречие сначала с p , а затем с q .
- Если текущая формула - $forall(X, p)$, для замещения X вводится новая переменная, правильное значение может быть получено позже, при унификации.
- Если текущая формула - $exists(X, p)$, для замещения X вводится новая константа.
- В противном случае, формула должна быть литералом, так что делается попытка унифицировать его противоречивым литералом.
- Если унификация не удалась, литерал добавляется в список литералов, а доказыватель переходит к обработке следующей формулы.

Нам необходима стратегия возврата, похожая на прологовскую: только если текущие связывания позволяют разрешить все оставшиеся цели, мы допускаем их. Можно было бы снова использовать списки, но давайте попробуем *продолжения* (*continuations*). Продолжение - это функция, передающаяся другой функции, которая может вызываться из последней «для выполнения остаточных вычислений». В нашем случае она принимает список с переменными и пытается разрешить оставшиеся цели при заданных связываниях. Следовательно вместо попыток явно разрешить оставшиеся цели, мы просто вызываем функцию-продолжение.

```

let rec prove fm unexp pl nl n cont i =
  if n < 0 then raise (error "No_proof") else
  match fm with
    Fn("&",[p;q]) ->
      prove p (q::unexp) pl nl n cont i
  | Fn("|",[p;q]) ->
      prove p unexp pl nl n
      (prove q unexp pl nl n cont) i
  | Fn("forall",[Var x; p]) ->
      let v = mkvar() in
      prove (subst [x,Var v] p) (unexp@[fm]) pl nl (n - 1) cont i
  | Fn("exists",[Var x; p]) ->
      let v = mkvar() in
      prove (subst [x,Fn(v,[[]]) p] unexp pl nl (n - 1) cont i
  | Fn("~",[t]) ->
      (try first (fun t' -> let i' = unify t t' i in
                    cont i')) pl
      with error _ ->
        prove (hd unexp) (tl unexp) pl (t::nl) n cont i
  | t ->
      (try first (fun t' -> let i' = unify t t' i in
                    cont i')) nl
      with error _ ->
        prove (hd unexp) (tl unexp) (t::pl) nl n cont i);;

```

И, наконец, доказыватель:

```
let prover =
  let rec prove_iter n t =
    try let insts = prove t [] [] [] n I [] in
        let globinsts = filter
          (fun (v,_) -> occurs_in v t) insts in
          n, globinsts
        with error _ -> prove_iter (n + 1) t in
  fun t -> prove_iter 0 (nnf_n(proc(parse_term t))));;
```

Так реализуется стратегия итеративного погружения. Доказыватель пытается найти доказательство с наименее общими связываниями переменных; если это удаётся, он возвращает их число и сами значения переменных.

Примеры

Вот несколько простых примеров, взятых из [Pelletier \(1986\)](#).

```
#let P1 = prover "p --> q <-> ~(q) --> ~(p)";;
P1 : int * (string * term) list = 0, []
#let P13 = prover "p | q & r <-> (p | q) & (p | r)";;
P13 : int * (string * term) list = 0, []
#let P16 = prover "(p --> q) | (q --> p)";;
P16 : int * (string * term) list = 0, []
#let P18 = prover "exists(Y,forall(X,p(Y)-->p(X)))";;
P18 : int * (string * term) list = 2, []
#let P19 = prover "exists(X,forall(Y,forall(Z,
  (p(Y)-->q(Z))-->p(X)-->q(X))))";;
P19 : int * (string * term) list = 6, []
```

Пример побольше:

```
#let P55 = prover
  "lives(agatha) & lives(butler) & lives(charles) &
  (killed(agatha,agatha) | killed(butler,agatha) |
  killed(charles,agatha)) &
  (forall(X,forall(Y,
    killed(X,Y) --> hates(X,Y) & ~(richer(X,Y)))) &
  (forall(X,hates(agatha,X)
    --> ~(hates(charles,X)))) &
  (hates(agatha,agatha) & hates(agatha,charles)) &
  (forall(X,lives(X) & ~(richer(X,agatha))
    --> hates(butler,X))) &
  (forall(X,hates(agatha,X) --> hates(butler,X))) &
  (forall(X,~(hates(X,agatha)) | ~(hates(X,butler))
    | ~(hates(X,charles))))
  --> killed(agatha,agatha)";;
P55 : int * (string * term) list = 6, []
```

Фактически доказыватель может работать детективом:

```
#let P55' = prover
  "lives(agatha) & lives(butler) & lives(charles) &
    (killed(agatha,agatha) | killed(butler,agatha) |
     killed(charles,agatha)) &
    (forall(X,forall(Y,
      killed(X,Y) --> hates(X,Y) & ~(richer(X,Y)))) &
     (forall(X,hates(agatha,X)
      --> ~(hates(charles,X)))) &
     (hates(agatha,agatha) & hates(agatha,charles)) &
     (forall(X,lives(X) & ~(richer(X,agatha))
      --> hates(butler,X))) &
     (forall(X,hates(agatha,X) --> hates(butler,X))) &
     (forall(X,~(hates(X,agatha)) | ~(hates(X,butler))
      | ~(hates(X,charles))))
    --> killed(X,agatha)";;
P55' : int * (string * term) list = 6, ["X", 'agatha']
```

Дальнейшее чтение

Символьное дифференцирование - классическое приложение для функциональных языков. Другие символьные операции также используются – [Davenport, Siret, and Tournier \(1988\)](#) содержит обзор того, что могут делать системы компьютерной алгебры и как они работают. В [Paulson \(1983\)](#) обсуждается стратегия упрощения, использованная нами в более общих условиях.

Синтаксический анализ с помощью функций высшего порядка - другой популярный пример. Похоже он давно уже вошёл в фольклор функционального программирования; первоначально он был рассмотрен в [Burge \(1975\)](#). Наши примеры основываются на [Paulson \(1991\)](#) и [Reade \(1989\)](#).

Первое определение «вычислимого вещественного числа» было дано в [Turing \(1936\)](#). Его подход основывался на десятичных расширениях, но позже потребовались изменения – ([Turing 1937](#)), по причинам, рассматриваемым далее в упражнениях. Подход к вещественной арифметике, осуществлённый нами, основывается на работе [Boehm, Cartwright, O'Donnel, and Riggle \(1986\)](#). Недавняя более эффективная реализация на CAML Light была описана в [Ménissier-Morain \(1994\)](#). Эта статья (на французском) содержит подробное доказательство корректности всех алгоритмов для элементарных трансцендентных функций. Альтернативное решение использует «дробно-линейные преобразования», см. [Potts \(1996\)](#).

Наш подход к прологовскому поиску и возврату, основанный на использовании списков и продолжений, достаточно стандартен. За подробностями о реализации Пролога обращайтесь, например, к [Boizumault \(1993\)](#), с подробностями о текущем использовании языка стоит ознакомиться по [Clocksin and Mellish \(1987\)](#). Детальное обсуждение продолжений, во многих их проявлениях, даётся в [Reynolds \(1993\)](#). Алгоритм унификации, рассмотренный в курсе прост и достаточно функционален, но не блещет эффективностью. За информацией о более быстрых императивных алгоритмах обращайтесь к [Martelli and Montanari \(1982\)](#). Наш доказыватель теорем основывается на lean^{TP} ([Beckert and Posegga 1995](#)). О другом важном методе доказывания теорем, для удобства основывающемся на прологовском стиле поиска, можно прочитать в ([Stickel 1988](#)).

Упражнения

1. Модифицируйте систему печати так, чтобы она учитывала ассоциативность операторов и не печатала избыточных скобок.
2. Правила дифференцирования для $1/g(x)$ и $f(x)/g(x)$ не корректны, когда $g(x) = 0$. В действительности большинство коммерческих систем компьютерной алгебры игнорирует это факт. Напишите новую версию `differentiate`, которая возвращает не только производную, но также и список условий, которые должны удовлетворяться, чтобы производная считалась корректной.
3. Запрограммируйте простую процедуру интегрирования. Достаточно реализовать основные правила интегрирования, хотя это не позволит взять любой интеграл.⁶
4. Прочитайте документацию на библиотеку `format`, и попробуйте реализовать форматированную печать многострочного вывода.
5. Что случится, если функции синтаксического анализатора - `term`, `atom` и прочие будут η -редуцированы удалением слова `input`? Что случится, если `precedence` будет последовательно η -редуцироваться с помощью удаления `input`?
6. На сколько хорошо анализатор с приоритетами, сгенерированный функцией `precedence`, обрабатывает различные операторы с одинаковым приоритетом? Как можно улучшить его работу? Можете ли вы усовершенствовать его так, чтобы запись допускала смешение лево- и правоассоциативных операторов?
7. Перепишите анализатор так, чтобы при ошибках разбора он возвращал информативные сообщения.
Представьте вещественное число с помощью функции, генерирующей первые
8. (*) ???TODO спросить у мамы)
9. Реализуйте мультиразрядную циклическую арифметику. Постройте алгоритм произведения, выполняющийся за $O(n^{\log_2 3})$ (??? времени памяти???), где n – общее число цифр в аргументах.
10. (*) Extend the real arithmetic routines для функций типа `exp` и `sin`.
11. Наша подготовка и приведение к негативной нормальной форме могут выполняться за время экспоненциальное от длины ввода, если ввод содержит множество вхождений операторов эквивалентности. Модифицируйте функции так, чтобы зависимость времени от числа операторов была линейной.
12. Измените преобразование `nnf_p` / `nnf_n` так, чтобы оно возвращало Сколемовскую нормальную форму.⁷

⁶В действительности, есть алгоритмы, позволяющие интегрировать произвольное алгебраическое выражение (не содержащее `sin`, `ln` и пр.) — подробности в [Davenport \(1981\)](#).

⁷Сколемовская нормальная форма рассматривается в большинстве курсов элементарной логики, например в [Enderton \(1972\)](#). TODO: вариант книги на русском

13. (*) Следуя, например, [Martelli and Montanari \(1982\)](#) реализуйте более эффективный алгоритм унификации.
14. (*) Реализуйте аналог Prolog Technology Theorem Prover.
([Stickel 1988](#))

Bibliography

- Aagaard, M. and Leeser, M. (1994) Verifying a logic synthesis tool in Nuprl: A case study in software verification. In v. Bochmann, G. and Probst, D. K. (eds.), *Computer Aided Verification: Proceedings of the Fourth International Workshop, CAV'92*, Volume 663 of *Lecture Notes in Computer Science*, Montreal, Canada, pp. 69–81. Springer Verlag.
- Abramsky, S. (1990) The lazy lambda-calculus. In Turner, D. A. (ed.), *Research Topics in Functional Programming*, Year of Programming series, pp. 65–116. Addison-Wesley.
- Adel'son-Vel'skii, G. M. and Landis, E. M. (1962) An algorithm for the organization of information. *Soviet Mathematics Doklady*, **3**, 1259–1262.
- Backus, J. (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, **21**, 613–641.
- Barendregt, H. P. (1984) *The Lambda Calculus: Its Syntax and Semantics*, Volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland.
- Barwise, J. (1989) Mathematical proofs of computer correctness. *Notices of the American Mathematical Society*, **7**, 844–851.
- Beckert, B. and Posegga, J. (1995) $\text{lean}T^A_P$: Lean, tableau-based deduction. *Journal of Automated Reasoning*, **15**, 339–358. Also available on the Web from <ftp://sonja.ira.uka.de/pub/posegga/LeanTaP.ps.Z>.
- Boehm, H. J., Cartwright, R., O'Donnel, M. J., and Riggle, M. (1986) Exact real arithmetic: a case study in higher order programming. In *Conference Record of the 1986 ACM Symposium on LISP and Functional Programming*, pp. 162–173. Association for Computing Machinery.
- Boizumault, P. (1993) *The implementation of Prolog*. Princeton series in computer science. Princeton University Press. Translated from 'Prolog: l'implantation' by A. M. Djambouliau and J. Fattouh.
- Boyer, R. S. and Moore, J. S. (1979) *A Computational Logic*. ACM Monograph Series. Academic Press.
- Burge, W. H. (1975) *Recursive Programming Techniques*. Addison-Wesley.
- Church, A. (1936) An unsolvable problem of elementary number-theory. *American Journal of Mathematics*, **58**, 345–363.

- Church, A. (1940) A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, **5**, 56–68.
- Church, A. (1941) *The calculi of lambda-conversion*, Volume 6 of *Annals of Mathematics Studies*. Princeton University Press.
- Clocksin, W. F. and Mellish, C. S. (1987) *Programming in Prolog* (3rd ed.). Springer-Verlag.
- Curry, H. B. (1930) Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, **52**, 509–536, 789–834.
- Davenport, J. H. (1981) *On the integration of algebraic functions*, Volume 102 of *Lecture Notes in Computer Science*. Springer-Verlag.
- Davenport, J. H., Siret, Y., and Tournier, E. (1988) *Computer algebra: systems and algorithms for algebraic computation*. Academic Press.
- Davis, M. D., Sigal, R., and Weyuker, E. J. (1994) *Computability, complexity, and languages: fundamentals of theoretical computer science* (2nd ed.). Academic Press.
- de Bruijn, N. G. (1972) Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, **34**, 381–392.
- DeMillo, R., Lipton, R., and Perlis, A. (1979) Social processes and proofs of theorems and programs. *Communications of the ACM*, **22**, 271–280.
- Dijkstra, E. W. (1976) *A Discipline of Programming*. Prentice-Hall.
- Enderton, H. B. (1972) *A Mathematical Introduction to Logic*. Academic Press.
- Frege, G. (1893) *Grundgesetze der Arithmetik begriffsschrift abgeleitet*. Jena. Partial English translation by Montgomery Furth in ‘The basic laws of arithmetic. Exposition of the system’, University of California Press, 1964.
- Girard, J.-Y., Lafont, Y., and Taylor, P. (1989) *Proofs and Types*, Volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.
- Gordon, A. D. (1994) *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press.
- Gordon, M. J. C. (1988) *Programming Language Theory and its Implementation: applicative and imperative paradigms*. Prentice-Hall International Series in Computer Science. Prentice-Hall.
- Gordon, M. J. C., Milner, R., and Wadsworth, C. P. (1979) *Edinburgh LCF: A Mechanised Logic of Computation*, Volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag.
- Henson, M. C. (1987) *Elements of functional languages*. Blackwell Scientific.

- Hindley, J. R. and Seldin, J. P. (1986) *Introduction to Combinators and λ -Calculus*, Volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press.
- Hudak, P. (1989) Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, **21**, 359–411.
- Huet, G. (1980) Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, **27**, 797–821.
- Kleene, S. C. (1935) A theory of positive integers in formal logic. *American Journal of Mathematics*, **57**, 153–173, 219–244.
- Lagarias, J. (1985) The $3x + 1$ problem and its generalizations. *The American Mathematical Monthly*, **92**, 3–23. Available on the Web as <http://www.cecm.sfu.ca/organics/papers/lagarias/index.html>.
- Landin, P. J. (1966) The next 700 programming languages. *Communications of the ACM*, **9**, 157–166.
- Lindemann, F. (1882) Über die Zahl π . *Mathematische Annalen*, **120**, 213–225.
- Mairson, H. G. (1990) Deciding ML typability is complete for deterministic exponential time. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, San Francisco, pp. 382–401. Association for Computing Machinery.
- Mairson, H. G. (1991) Outline of a proof theory of parametricity. In Hughes, J. (ed.), *1991 ACM Symposium on Functional Programming and Computer Architecture*, Volume 523 of *Lecture Notes in Computer Science*, Harvard University, pp. 313–327.
- Martelli, A. and Montanari, U. (1982) An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, **4**, 258–282.
- Mauny, M. (1995) Functional programming using CAML Light. Available on the Web from <http://pauillac.inria.fr/caml/tutorial/index.html>.
- Ménissier-Morain, V. (1994) *Arithmétique exacte, conception, algorithmique et performances d'une implémentation informatique en précision arbitraire*. Thèse, Université Paris 7.
- Michie, D. (1968) “Memo” functions and machine learning. *Nature*, **218**, 19–22.
- Milner, R. (1978) A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, **17**, 348–375.
- Mycroft, A. (1981) Abstract interpretation and optimising transformations of applicative programs. Technical report CST-15-81, Computer Science Department, Edinburgh University, King’s Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK.
- Neumann, P. G. (1995) *Computer-related risks*. Addison-Wesley.
- Oppen, D. (1980) Prettyprinting. *ACM Transactions on Programming Languages and Systems*, **2**, 465–483.

- Paulson, L. C. (1983) A higher-order implementation of rewriting. *Science of Computer Programming*, **3**, 119–149.
- Paulson, L. C. (1991) *ML for the Working Programmer*. Cambridge University Press.
- Pelletier, F. J. (1986) Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, **2**, 191–216. Errata, JAR 4 (1988), 235–236.
- Peterson, I. (1996) *Fatal Defect : Chasing Killer Computer Bugs*. Arrow.
- Potts, P. (1996) Computable real arithmetic using linear fractional transformations. Unpublished draft for PhD thesis, available on the Web as <http://theory.doc.ic.ac.uk/~pjp/pub/phd/draft.ps.gz>.
- Raphael, B. (1966) The structure of programming languages. *Communications of the ACM*, **9**, 155–156.
- Reade, C. (1989) *Elements of Functional Programming*. Addison-Wesley.
- Reynolds, J. C. (1993) The discoveries of continuations. *Lisp and Symbolic Computation*, **6**, 233–247.
- Robinson, J. A. (1994) Logic, computers, Turing and von Neumann. In Furukawa, K., Michie, D., and Muggleton, S. (eds.), *Machine Intelligence 13*, pp. 1–35. Clarendon Press.
- Schönfinkel, M. (1924) Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, **92**, 305–316. English translation, ‘On the building blocks of mathematical logic’ in ?), pp. 357–366.
- Schwichtenberg, H. (1976) Definierbare Funktionen im λ -Kalkül mit Typen. *Arkiv für mathematische Logik und Grundlagenforschung*, **17**, 113–114.
- Stickel, M. E. (1988) A Prolog Technology Theorem Prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, **4**, 353–380.
- Turing, A. M. (1936) On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society (2)*, **42**, 230–265.
- Turing, A. M. (1937) Correction to Turing (1936). *Proceedings of the London Mathematical Society (2)*, **43**, 544–546.
- Whitehead, A. N. (1919) *An Introduction to Mathematics*. Williams and Norgate.
- Whitehead, A. N. and Russell, B. (1910) *Principia Mathematica (3 vols)*. Cambridge University Press.
- Winskel, G. (1993) *The formal semantics of programming languages: an introduction*. Foundations of computing. MIT Press.
- Wittgenstein, L. (1922) *Tractatus Logico-Philosophicus*. Routledge & Kegan Paul.