Классика программирования

Жиль Довек, Жан-Жак Леви

Введение в теорию языков программирования

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Жиль Довек, Жан-Жак Леви

Введение в теорию языков программирования

ДМК Пресс
издательство
Москва, 2013

Introduction to the Theory of Programming Languages

Gilles Dowek, Jean-Jacques Lévy

Springer

Введение в теорию языков программирования

Жиль Довек, Жан-Жак Леви

Перевод с английского
В. Н. Брагилевского и А. М. Пеленицына

Издание рекомендовано в качестве учебного пособия
для студентов технических вузов

ДМК ПРЕСС
издательство

Москва, 2013

УДК 004.42
ББК 32.973-018
Д58

Д 58 **Довек Жиль, Леви Жан-Жак**
Введение в теорию языков программирования — Пер. с англ. —
М.: ДМК Пресс, 2013. — 134 с.: ил.
ISBN 978-5-94074-913-4

Языки программирования от Фортрана и Кобола до Caml и Java играют ключевую роль в управлении сложными компьютерными системами. Книга «Введение в теорию языков программирования» представляет читателю средства, необходимые для проектирования и реализации подобных языков. В ней предлагается единый подход к различным формализмам для определения языков программирования — операционной и денотационной семантике. Особое внимание при этом уделяется способам задания отношений между тремя объектами: программой, входным значением и результатом. Эти формализмы демонстрируются на примере таких типичных элементов языков программирования, как функции, рекурсия, присваивание, записи и объекты. При этом показывается, что теория языков программирования состоит не в последовательном изучении самих языков один за другим, а строится вокруг механизмов, входящих в различные языки. Изучение таких механизмов в книге приводит к разработке вычислителей, интерпретаторов и компиляторов, а также к реализации алгоритмов вывода типов для учебных языков.

УДК 004.42
ББК 32.973-018

ISBN 978-0-85729-075-5 (англ.)
ISBN 978-5-94074-913-4 (рус.)

Оглавление

От переводчиков

Теория языков программирования является важным разделом современной теоретической информатики, она активно развивается в работах западных специалистов, ей посвящаются большие конференции, статьи по этой тематике публикуются в ведущих научных журналах. К сожалению, эта теория долгое время оставалась на обочине советской, а затем российской науки, что, видимо, привело к дефициту русскоязычной литературы, в которой излагались бы её основы. Тем не менее, хотелось бы указать несколько исключений.

В первую очередь следует упомянуть сборник переводов под общим названием «Математическая логика в программировании» [1], изданный в 1991 году. В нём были опубликованы статьи таких известных учёных как Дана Скотт, Роджер Хиндли, Саймон Пейтон Джонс и других. Большой вклад в доведение теории языков программирования до российского читателя много позже внесла публикация перевода обширной научной монографии Джона Митчелла «Основания языков программирования» [2], выполненного под научной редакцией Н. Н. Непейводы. Наконец, благодаря труду энтузиастов был издан перевод книги Бенджамина Пирса «Типы в языках программирования» [3].

Настоящая книга представляет собой введение в теорию языков программирования, по содержанию и стилю изложения намного более доступное, чем любая из перечисленных книг. Авторам удалось вместить в очень небольшой объём самые необходимые сведения, удачно соединив вопросы операционной и денотационной семантики программ и теорию типов. В книге также нашлось место достаточному числу упражнений, что исключительно важно для начинающих. В результате данное издание можно рекомендовать для факультативов или семинаров на младших курсах университетов, а также для самостоятельного изучения. Упомянутые выше книги могли бы составить основу для более глубокого знакомства с темой на старших курсах, что обеспечило бы теории языков программирования более достойное место в университетских курсах, чем мы видим сейчас в нашей стране.

Несмотря на выход указанных изданий, в литературе по данному предмету остаётся несколько существенных пробелов, обозначим два из них. В теории языков программирования можно выделить два крупных раздела: теория типов и семантика языков программирования. Упомянутая выше книга Б. Пирса, а верней её перевод, даёт достаточно полное представление о теории типов. К сожалению, нам неизвестно перевода или оригинальной сравнимой по уровню работы на русском языке, посвящённой семантике языков программирования. Достойными кандидатами на эту роль видятся книги Карла Гюнтера или Глинна Винскеля, приведённые в авторском списке литературы в конце книги (пункты 3 и 14 соответственно), если бы они были изданы на русском языке.

Для глубокого изучения теории языков программирования совершенно необходимо знакомство с лямбда-исчислением, которое вполне можно считать тем самым «единственным универсальным» языком программирования, упомянутым авторами во введении к данной книге, но только для специалистов по теории языков программирования. Оно вводится в том или ином объёме в книгах Б. Пирса и Д. Митчелла, но (весьма разумно) обойдено стороной в настоящем издании. Тем не менее, полезно было бы иметь русскоязычное издание, посвящённое целиком этому предмету. Стоит отметить, что в издательстве «Мир» ещё в 1985 году выходил перевод энциклопедической книги выдающегося голландского учёного и специалиста в этой области Хенка Барендрегта «Ламбда-исчисление. Его синтаксис и семантика» [4]. Однако эта работа целиком посвящена бестиповому лямбда-исчислению, в то время как для современной теории языков программирования более важным представляется лямбда-исчисление с типами, великолепно описанное в доступной форме университетского учебника, например, Роджером Хиндли и Джонатаном Селдином [5]. Пример того, как более глубокое изучение вопросов типизации языков программирования часто проводится в рамках лямбда-исчисления, дают пятая и шестая главы настоящей книги, в которых приведены два различных подхода к типизации. Эти два подхода по существу соответствуют «типизации по Чёрчу» и «типизации по Карри», детально рассматриваемым в книге Р. Хиндли и Дж. Селдина.

Ещё один компонент общего образования в информатике, который используется в большинстве книг по теории языков программирования, это функциональное программирование. Прекрасный обзор переводной и оригинальной литературы по этому предмету содержится в статье [6]. Заметим лишь, что в данной книге, как и в издании труда Б. Пирса, для иллюстрации материала используется язык программирования ML, относительно непопулярный в российском академическом сообществе. Нетрудно

выяснить, что в большинстве университетских курсов в России для обсуждения функционального программирования предлагается старейший язык Лисп, которому заметно больше повезло с переводными изданиями в нашей стране (а также их тиражами). Западные университеты довольно давно стали отказываться от Лиспа и его диалектов в пользу ML, а в последнее время популярность завоёвывает более современный язык функционального программирования Haskell, отголоски этого процесса можно видеть и в некоторых отечественных университетах. По всей видимости, если в России и произойдёт смещение интересов университетских преподавателей от Лиспа, то центром притяжения для них станет скорее более актуальный язык Haskell, ML же останется для многих своеобразной «потерянной главой». Этот факт, тем не менее, вряд ли сможет помешать знакомству с теорией языков программирования, потому что в соответствующих книгах обычно используются самые базовые, интуитивно понятные возможности этого языка. Стоит лишь иметь в виду более общее соображение: некоторый опыт работы с функциональными языками весьма полезен для знакомства с темой, которой посвящена настоящая книга.

Малое число переводов неизбежно приводит к терминологическим трудностям, и мы хотели бы сделать несколько замечаний по этому поводу. Отметим, что термин *confluency* мы переводим как *свойство слияния* (в отличие от *конфлюэнтности* и *сходимости*, как в переводах книг Б. Пирса и Д. Митчелла соответственно). Следуя переводу книги Митчелла и избегая при этом дословного перевода использованного в книге термина *standardisation*, свойство гарантированной достижимости нормальной формы терма, при условии её существования, редукцией с вызовом по имени мы называем *полнотой*. Сложный для перевода и обычно либо транслитерируемый, либо переводимый как *отложенный вызов* термин *thunk* у нас переведён как *задумка*, тогда как *редекс*, прочно закрепившийся в русскоязычной литературе ещё с перевода книги Х. Барендрегта, остался транслитерированным. Читателю также будет полезно иметь в виду, что авторы считают *ноль* натуральным числом.

В. Н. Брагилевский и А. М. Пеленицын,
факультет математики, механики и компьютерных наук
Южного федерального университета

Литература

1. Математическая логика в программировании: сборник статей 1980—1988 гг. М.: Мир, 1991. — 408 с.

2. Митчелл Дж. Основания языков программирования. М.–Ижевск: НИЦ «Регулярная и хаотическая динамика», 2010. — 720 с.

3. Пирс Б. Типы в языках программирования. М.: Лямбда пресс, Добросвет, 2012. — 680 с.

4. Барендрегт Х. Ламбда-исчисление. Его синтаксис и семантика. М.: Мир, 1985. — 606 с.

5. Hindley J.R., Seldin J.P. Lambda-calculus and Combinators, an Introduction. Cambridge University Press, 2008. P. 345.

6. Отт А. Обзор литературы о функциональном программировании // Практика функционального прграммирования, вып. 1. 2009. С. 93–114.

Что называют теорией языков программирования?

Человечество пока ещё довольно далеко от создания единственного универсального языка программирования. Новые языки появляются почти каждый день, к уже существующим добавляются новые возможности. Улучшения в языках программирования служат созданию более надёжных программ, сокращают время разработки и упрощают сопровождение. Улучшения необходимы и для удовлетворения новым требованиям, таким как разработка параллельных, распределённых или мобильных приложений.

Определение языка программирования начинается с описания его синтаксиса. Что предпочесть, `x := 1` или `x = 1`? Нужно ли ставить скобки после `if` или нет? И вообще, какую последовательность символов можно считать программой? Для ответа на подобные вопросы имеется полезный инструмент: формальная грамматика. Используя грамматику, можно точно описать синтаксис языка программирования, что, в свою очередь, помогает создавать программы проверки синтаксической корректности других программ.

Однако даже строгое определение синтаксически корректной программы не позволяет предсказать, что именно случится после её запуска. Определяя язык программирования, необходимо также описать его семантику, то есть ожидаемое поведение программы во время исполнения. Два языка могут иметь одинаковый синтаксис, но различную семантику.

Приведём пример того, что неформально понимают под семантикой. Вычисление значения функции обычно объясняют следующим образом. «*Для вычисления результата* V *выражения, записанного в форме* $f\ e_1\ \ldots\ e_n$*, где символ* f *обозначает функцию, определяемую выражением* $f\ x_1\ \ldots\ x_n = e'$*, необходимо: во-первых, вычислить значения* $W_1, \ldots, W_n$ *аргументов* $e_1, \ldots, e_n$*, затем связать эти значения с переменными* $x_1, \ldots, x_n$*, и, наконец, вычислить выражение* e'*. Полученное в итоге значение* V *и есть результат вычисления*».

Такое объяснение семантики, выраженное естественным языком (русским, в данном случае), конечно, даст нам представление о том, что произойдёт во время исполнения программы, но насколько точным это представление окажется? Рассмотрим, к примеру, программу:

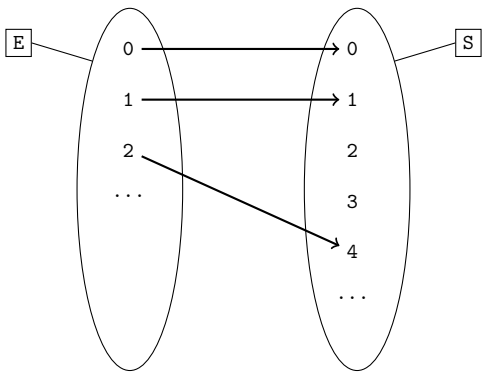
```
f x y = x
g z = (n = n + z; n)
n = 0; print (f (g 2) (g 7))
```

В зависимости от способа интерпретации данного выше объяснения можно установить, что результатом программы будет либо значение 2, либо значение 9. Причина в том, что объяснение на естественном языке не уточняет, следует ли вычислять `g 2` до или после `g 7`, тогда как в рассматриваемой программе порядок вычисления важен. В объяснении следовало бы сказать: «аргументы $\mathtt{e}_1, \ldots, \mathtt{e}_n$ вычисляются, *начиная с* $\mathtt{e}_1$», или наоборот, «*начиная с* $\mathtt{e}_n$».

Два программиста, прочитав неоднозначное объяснение, могут понять его по-разному. Хуже того, разработчики компиляторов языка могут выбрать различные соглашения. Тогда одна и та же программа будет давать разные результаты в зависимости от используемого компилятора.

Хорошо известно, что естественные языки слишком неточны для описания синтаксиса языка программирования, вместо них следует использовать формальные языки. В случае семантики ситуация аналогична, для её описания также необходим некий формальный язык.

Так что же такое семантика программы? Возьмём, к примеру, программу `p`, которая запрашивает целое число, вычисляет его квадрат и отображает полученный результат. Чтобы описать поведение этой программы, нам понадобится ввести отношение `R` между входными значениями и соответствующими им результатами.



Теперь можно сказать, что семантикой этой программы является отношение `R` между элементами множества входных значений `E` и элементами множества выходных значений `S`, то есть некоторое подмножество декартова произведения $\mathtt{E} \times \mathtt{S}$.

Следовательно, семантика программы является бинарным отношением. В свою очередь, семантика языка программирования это тернарное отношение: «программа `p` с входным значением `e` возвращает выходное значение `s`». Обозначим это отношение следующим образом: `p, e` $\hookrightarrow$ `s`. Программа `p` и вход `e` доступны до начала выполнения программы. Зачастую эти два элемента объединяются в *терм* `p e`, и семантика языка приписывает значение этому терму. В этом случае семантика языка оказывается бинарным отношением `t` $\hookrightarrow$ `s`.

Теперь для выражения семантики языка программирования нам требуется язык, позволяющий описывать отношения.

Если семантика программы является функциональным отношением, то есть для каждого входного значения существует не более одного выходного, будем говорить, что программа является *детерминированной*.

Примерами недетерминированных программ являются компьютерные игры, поскольку для того, чтобы игра доставляла удовольствие, необходим элемент случайности. Язык называется детерминированным, если все программы, которые можно написать на этом языке, детерминированы, или, что то же самое, если его семантика является функциональным отношением. В этом случае семантику можно определить не описанием отношений, а с помощью некоторого языка описания функций.

Благодарности

Авторы хотели бы поблагодарить Жерара Асайа (Gérard Assayag), Антонио Бучарелли (Antonio Bucciarelli), Роберто Ди Козмо (Roberto Di Cosmo), Ксавье Леруа (Xavier Leroy), Дейва МакКуина (Dave MacQueen), Люка Мароже (Luc Maranget), Мишеля Мони (Michel Mauny), Франсуа Потье (François Pottier), Дидье Реми (Didier Rémy), Алана Шмитта (Alan Schmitt), Элоди-Джейн Зим (Élodie-Jane Sims) и Веронику Вигье Донзе-Гуж (Véronique Viguié Donzeau-Gouge).

Глава 1. Термы и отношения

1.1. Индуктивные определения

Поскольку семантика языка программирования представляет собой отношение, мы начнём с описания некоторых средств задания множеств и отношений.

Самым простым средством является *явное определение*. Мы можем, к примеру, явно определить функцию, умножающую свой аргумент на 2: $\mathtt{x} \mapsto \mathtt{2 * x}$, множество чётных чисел: $\{\mathtt{n} \in \mathbb{N} \mid \exists \mathtt{p} \in \mathbb{N}\ \ \mathtt{n = 2 * p}\}$, или отношение делимости: $\{\mathtt{(n, m)} \in \mathbb{N}^2 \mid \exists \mathtt{p} \in \mathbb{N}\ \ \mathtt{n = p * m}\}$. Однако подобных явных определений недостаточно для описания всех требующихся нам объектов. Вторым средством для задания множеств и отношений является *индуктивное определение*. Оно основывается на простом факте — теореме о неподвижной точке.

1.1.1. Теорема о неподвижной точке

Рассмотрим отношение порядка (упорядочение) $\leqslant$, то есть рефлексивное, антисимметричное и транзитивное отношение на множестве $\mathtt{E}$, и пусть для элементов последовательности $\mathtt{u_0, u_1, u_2, \ldots}$ имеет место цепочка неравенств $\mathtt{u_0} \leqslant \mathtt{u_1} \leqslant \mathtt{u_2} \leqslant \ldots$, то есть последовательность является возрастающей. Элемент $\mathtt{l}$ множества $\mathtt{E}$ называется *пределом* последовательности $\mathtt{u_0, u_1, u_2, \ldots}$, если он представляет собой наименьшую верхнюю грань множества $\{\mathtt{u_0, u_1, u_2, \ldots}\}$, то есть, если:

(1) $\mathtt{u_i} \leqslant \mathtt{l}$ для всех $\mathtt{i}$;

(2) из того, что $\mathtt{u_i} \leqslant \mathtt{l'}$ для всех $\mathtt{i}$, следует $\mathtt{l} \leqslant \mathtt{l'}$.

Если предел существует, то он единственен и обозначается $\mathtt{lim_i\, u_i}$.

Отношение порядка $\leqslant$ называется *слабо полным*, если каждая возрастающая последовательность имеет предел.

Обычное отношение порядка на отрезке вещественных чисел $\mathtt{[0, 1]}$ даёт пример слабо полного порядка. Кроме того, это отношение имеет наименьший элемент 0. В то же время обычное отношение порядка на $\mathbb{R}^+$ не является слабо полным, так как возрастающая последовательность $\mathtt{1, 2, 3, \ldots}$ не имеет предела.

Рассмотрим произвольное множество A. Отношение включения $\subseteq$ на множестве $\wp$(A) всех подмножеств множества A является ещё одним примером слабо полного порядка. Пределом возрастающей последовательности $U_0, U_1, U_2, \ldots$ является множество $\bigcup_{i\in\mathbb{N}} U_i$. Кроме того, это отношение имеет наименьший элемент $\varnothing$.

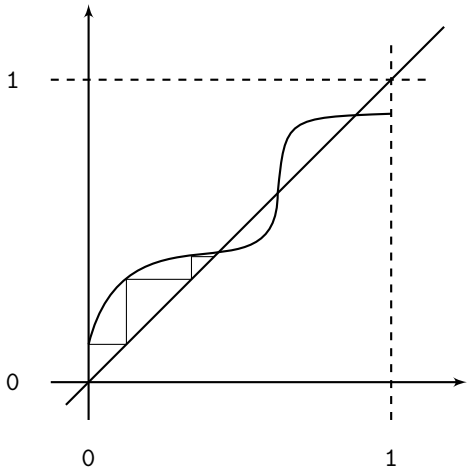
Пусть функция f действует из E в E. Она называется *возрастающей*, если

$$x \leqslant y \quad \Longrightarrow \quad f\,x \leqslant f\,y.$$

Функция f называется *непрерывной*, если в дополнение к этому для любой возрастающей последовательности выполнено:

$$\lim_i (f\,u_i) = f(\lim_i u_i).$$

Теорема 1 (первая теорема о неподвижной точке)**.** *Пусть* $\leqslant$ *— слабо полное отношение порядка на множестве* E*, которое имеет наименьший элемент* m*. Пусть функция* f *действует из* E *в* E*. Если* f *непрерывна, то* $p = \lim_i (f^i m)$ *является наименьшей неподвижной точкой* f*.*



Доказательство. Во-первых, поскольку m это наименьший элемент множества E, то $m \leqslant f\,m$. Функция f возрастает, поэтому $f^i m \leqslant f^{i+1} m$. Так как последовательность $f^i m$ возрастает, она имеет предел. Последовательность $f^{i+1} m$ также имеет предел, равный p, и потому $p = \lim_i (f(f^i m)) =$

$\mathtt{f}(\lim_{\mathtt{i}}(\mathtt{f}^{\mathtt{i}}\mathtt{m})) = \mathtt{f}\,\mathtt{p}$. Более того, $\mathtt{p}$ является наименьшей неподвижной точкой, потому что если $\mathtt{q}$ это другая неподвижная точка, то $\mathtt{m} \leqslant \mathtt{q}$ и $\mathtt{f}^{\mathtt{i}}\mathtt{m} \leqslant \mathtt{f}^{\mathtt{i}}\mathtt{q} = \mathtt{q}$ (так как $\mathtt{f}$ возрастающая). А значит, $\mathtt{p} = \lim_{\mathtt{i}}(\mathtt{f}^{\mathtt{i}}\mathtt{m}) \leqslant \mathtt{q}$. ■

Вторая теорема о неподвижной точке утверждает существование неподвижной точки для возрастающих функций, которые могут не быть непрерывными, в случае, когда порядок удовлетворяет более сильному требованию.

Отношение порядка $\leqslant$ на множестве $\mathtt{E}$ называется *сильно полным*, если каждое подмножество $\mathtt{A} \subseteq \mathtt{E}$ имеет наименьшую верхнюю грань $\sup\mathtt{A}$.

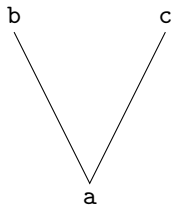
Обычное отношение порядка на отрезке $[0,1]$ даёт пример сильно полного отношения порядка. Одновременно обычное упорядочение на $\mathbb{R}^+$ не является сильно полным, потому что само множество $\mathbb{R}^+$ не имеет верхней грани.

Рассмотрим произвольное множество $\mathtt{A}$. Отношение включения $\subseteq$ на множестве $\wp(\mathtt{A})$ всех подмножеств $\mathtt{A}$ представляет собой другой пример сильно полного отношения порядка. Наименьшей верхней гранью множества $\mathtt{B}$ является множество $\bigcup_{\mathtt{C}\in\mathtt{B}}\mathtt{C}$.

Упражнение 1.1.

(1) Покажите, что любое сильно полное отношение порядка слабо полно.

(2) Является ли данное упорядочение слабо полным? Сильно полным?



Заметим, что если упорядочение $\leqslant$ на множестве $\mathtt{E}$ сильно полно, то любое подмножество $\mathtt{A} \subseteq \mathtt{E}$ имеет наибольшую нижнюю грань $\inf\mathtt{A}$. Действительно, пусть $\mathtt{A}$ является подмножеством $\mathtt{E}$, обозначим $\mathtt{B} = \{\mathtt{y} \in \mathtt{E} \mid \forall \mathtt{x} \in \mathtt{A}\ \ \mathtt{y} \leqslant \mathtt{x}\}$ множество нижних граней $\mathtt{A}$ и $\mathtt{l}$ — наименьшую верхнюю грань $\mathtt{B}$. По определению $\mathtt{l}$ является верхней гранью множества $\mathtt{B}$:

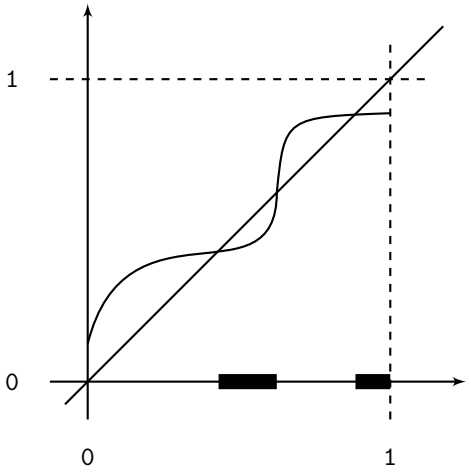
$$\forall \mathtt{y} \in \mathtt{B}\ \ \mathtt{y} \leqslant \mathtt{l},$$

это и наименьшая такая грань:

$$(\forall \mathtt{y} \in \mathtt{B}\ \ \mathtt{y} \leqslant \mathtt{l}') \Longrightarrow \mathtt{l} \leqslant \mathtt{l}'.$$

Легко показать, что l является наибольшей нижней гранью множества A. Действительно, если x ∈ A, то x является верхней гранью для B, и, так как l это наименьшая верхняя грань, то l ⩽ x. Таким образом, l является нижней гранью A. Чтобы показать, что она наибольшая, достаточно заметить, что если m — другая нижняя грань A, то m ∈ B и, значит, m ⩽ l.

Теорема 2 (вторая теорема о неподвижной точке)**.** *Пусть* ⩽ *— сильно полное упорядочение на множестве* E*. Функция* f *действует из* E *в* E*. Если* f *возрастает, то* p = inf{c | f c ⩽ c} *является наименьшей неподвижной точкой* f*.*



Доказательство. Пусть C = {c | f c ⩽ c} и c ∈ C. Тогда p ⩽ c, потому что p это нижняя грань C. Так как функция f возрастает, то f p ⩽ f c. Кроме того, f c ⩽ c, потому что c это элемент C, и по транзитивности: f p ⩽ c.

Элемент f p меньше, чем любой элемент C, а значит, меньше или равен наибольшей нижней грани: f p ⩽ p. Вследствие возрастания f получаем f(f p) ⩽ f p, поэтому f p принадлежит C, и, так как p это нижняя грань C, заключаем, что p ⩽ f p. По антисимметричности: p = f p.

Наконец, по определению все неподвижные точки f принадлежат C и, таким образом, превосходят c. ■

1.1.2. Индуктивные определения

Теперь мы увидим, как теоремы о неподвижных точках могут быть использованы для определения множеств и отношений.

Пусть A — произвольное множество, функция f действует из A^n в A и E — подмножество A. Множество E *замкнуто* относительно действия

функции $\mathtt{f}$, если для любых $\mathtt{a_1}, \ldots, \mathtt{a_n}$ из $\mathtt{E}$ элемент $\mathtt{f\,a_1 \ldots a_n}$ также принадлежит $\mathtt{E}$. Например, множество чётных чисел замкнуто относительно действия функции $\mathtt{n} \mapsto \mathtt{n+2}$.

Пусть $\mathtt{A}$ — произвольное множество. Индуктивное определение подмножества $\mathtt{E} \subseteq \mathtt{A}$ это семейство частичных функций: $\mathtt{f_1}$ из $\mathtt{A}^{\mathtt{n_1}}$ в $\mathtt{A}$, $\mathtt{f_2}$ из $\mathtt{A}^{\mathtt{n_2}}$ в $\mathtt{A}$ и т. д. Множество $\mathtt{E}$ определяется как наименьшее замкнутое относительно действия функций $\mathtt{f_1}$, $\mathtt{f_2}$, ... подмножество $\mathtt{A}$.

Например, подмножество $\mathbb{N}$, которое содержит все чётные числа, определяется индуктивно числом 0 — то есть функцией из $\mathbb{N}^0$ в $\mathbb{N}$, которая возвращает значение 0 — и функцией из $\mathbb{N}$ в $\mathbb{N}$: $\mathtt{n} \mapsto \mathtt{n+2}$. Подмножество $\{\mathtt{a,b,c}\}^*$, состоящее из слов вида $\mathtt{a^n bc^n}$, определяется индуктивно словом $\mathtt{b}$ и функцией $\mathtt{m} \mapsto \mathtt{amc}$. Вообще, любая контекстно-свободная грамматика может быть определена как индуктивное множество. В логике множество теорем является подмножеством всех утверждений, которое индуктивно определено аксиомами и правилами вывода.

Функции $\mathtt{f_1, f_2, \ldots}$ называются *правилами*. Вместо записи правила в виде $\mathtt{x_1 \ldots x_n} \mapsto \mathtt{t}$ мы будем использовать следующую нотацию:

$$\frac{\mathtt{x_1 \ldots x_n}}{\mathtt{t}}.$$

Например, множество чётных чисел определяется правилами

$$\frac{}{\mathtt{0}}, \qquad \frac{\mathtt{n}}{\mathtt{n+2}}.$$

Пусть $\mathtt{P}$ обозначает множество чётных чисел. Иногда мы будем записывать правила так:

$$\frac{}{\mathtt{0 \in P}}, \qquad \frac{\mathtt{n \in P}}{\mathtt{n+2 \in P}}.$$

Иногда, чтобы определить язык индуктивно, мы будем использовать нотацию, позаимствованную из теории формальных языков, где, к примеру, множество слов вида $\mathtt{a^n bc^n}$ задаётся так:

$$\begin{aligned}\mathtt{m} &= \mathtt{b}\\ &\mid \mathtt{amc}\end{aligned}$$

Чтобы продемонстрировать существование наименьшего множества $\mathtt{A}$, замкнутого относительно действия функций $\mathtt{f_1, f_2, \ldots}$, определим функцию $\mathtt{F}$ из $\wp(\mathtt{A})$ в $\wp(\mathtt{A})$:

$$\mathtt{F\,C} = \{\mathtt{x} \in \mathtt{A} \mid \exists \mathtt{i}\ \ \exists \mathtt{y_1 \ldots y_{n_i}} \in \mathtt{C}\ \ \mathtt{x} = \mathtt{f_i\,y_1 \ldots y_{n_i}}\}.$$

Подмножество $C \subseteq A$ замкнуто относительно действия функций $f_1, f_2, \ldots$, если и только если $FC \subseteq C$.

Функция F, очевидно, возрастает, то есть, если $C \subseteq C'$, то $FC \subseteq FC'$. Более того, она непрерывна: если $C_0 \subseteq C_1 \subseteq C_2 \subseteq \ldots$, то $F(\bigcup_j C_j) = \bigcup_j (F\,C_j)$. Действительно, если элемент $x \in A$ принадлежит $F(\bigcup_j C_j)$, то существует индекс i и элементы $y_1 \ldots y_{n_i} \in \bigcup_j C_j$, такие что $x = f_i\, y_1 \ldots y_{n_i}$. Каждый из этих элементов лежит в одном из C_j. Так как последовательность возрастает, то все элементы принадлежат некоторому наибольшему множеству C_k. Следовательно, $x \in F\,C_k$, а также $x \in \bigcup_j (F\,C_j)$. Наоборот, если $x \in \bigcup_j (F\,C_j)$, то он принадлежит некоторому $F\,C_k$, и, значит, существует номер i и элементы $y_1 \ldots y_{n_i} \in C_k$, такие что $x = f_i\, y_1 \ldots y_{n_i}$. Элементы $y_1 \ldots y_{n_i}$ принадлежат $\bigcup_j C_j$, а значит, $x \in F(\bigcup_j C_j)$.

Множество E определяется как наименьшая неподвижная точка функции F. Это наименьшее множество, удовлетворяющее свойству $FE = E$, и, в соответствии со второй теоремой о неподвижной точке, наименьшее множество, удовлетворяющее свойству $FE \subseteq E$. Таким образом, это наименьшее замкнутое относительно действия функций f_1, f_2, ... множество.

Множество чётных чисел не единственное подмножество $\mathbb{N}$, содержащее 0 и замкнутое относительно действия функции $n \mapsto n+2$, (само множество $\mathbb{N}$, к примеру, также удовлетворяет этим свойствам), но наименьшее из таковых. Оно может быть определено как пересечение всех таких множеств. Вторая теорема о неподвижной точке позволяет нам обобщить это наблюдение и определить E как пересечение всех замкнутых относительно действия функций f_1, f_2, ... множеств.

Первая теорема о неподвижной точке показывает, что элемент x принадлежит E в том и только том случае, когда существует некоторый номер k, такой что $x \in F^k\varnothing$. То есть, если существует функция f_i, такая что $x = f_i\, y_1 \ldots y_{n_i}$, где $y_1, \ldots, y_{n_i} \in F^{k-1}\varnothing$. С помощью итерации, то есть индукцией по k, мы можем показать, что элемент $x \in A$ тогда и только тогда лежит в E, когда существует дерево, узлы которого помечены элементами множества A, корень помечен x, и если узел помечен c, то его дети помечены $d_1, \ldots, d_n$, такими что для некоторого правила f имеем: $c = f\, d_1 \ldots d_n$. Такое дерево называется *выводом* x. Понятие вывода обобщает понятие доказательства в логике. Мы можем определить множество E как набор элементов $x \in A$, для которых существует вывод.

Мы будем использовать специальную нотацию для выводов. Во-первых, корень дерева будет расположен внизу, а листья наверху. Далее, мы будем писать черту над каждым узлом дерева, а над этой чертой — детей этого узла.

Число 8, к примеру, принадлежит множеству чётных чисел, как по-

казывает следующий вывод:

$$\dfrac{\dfrac{\dfrac{\dfrac{0}{2}}{4}}{6}}{8}$$

Если обозначить через $\mathtt{P}$ множество чётных чисел, можно записать вывод иначе:

$$\dfrac{\dfrac{\dfrac{\dfrac{0 \in \mathtt{P}}{2 \in \mathtt{P}}}{4 \in \mathtt{P}}}{6 \in \mathtt{P}}}{8 \in \mathtt{P}}$$

1.1.3. Структурная индукция

Индуктивные определения подсказывают метод проведения доказательств. Если свойство наследуется, то есть, если каждый раз, когда оно выполнено для $\mathtt{y}_1, \ldots, \mathtt{y}_{\mathtt{n}_i}$, оно также выполнено и для $\mathtt{f}_\mathtt{i}\ \mathtt{y}_1 \ldots \mathtt{y}_{\mathtt{n}_i}$, то можно заключить, что оно присуще всем элементам $\mathtt{E}$.

Обосновать это можно, используя вторую теорему о неподвижной точке и заметив, что подмножество $\mathtt{P} \subseteq \mathtt{A}$, содержащее все элементы, которые удовлетворяют данному свойству, замкнуто относительно действия функций $\mathtt{f}_\mathtt{i}$ и, таким образом, включает $\mathtt{E}$. Другой способ обоснования — использовать первую теорему о неподвижной точке и показать индукцией по $\mathtt{k}$, что все элементы $\mathtt{F}^\mathtt{k}\varnothing$ удовлетворяют рассматриваемому свойству.

1.1.4. Рефлексивно-транзитивное замыкание отношения

Рефлексивно-транзитивное замыкание отношения является примером индуктивного определения. Если $\mathtt{R}$ — бинарное отношение на множестве $\mathtt{A}$, мы можем индуктивно определить другое отношение $\mathtt{R}^*$, называемое рефлексивно-транзитивным замыканием $\mathtt{R}$:

$$\dfrac{}{\mathtt{x}\,\mathtt{R}^*\,\mathtt{y}} \text{ если } \mathtt{x}\,\mathtt{R}\,\mathtt{y},$$

$$\dfrac{}{\mathtt{x}\,\mathtt{R}^*\,\mathtt{x}},$$

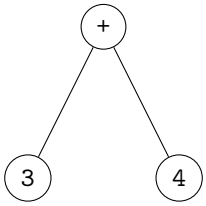
$$\dfrac{\mathtt{x}\,\mathtt{R}^*\,\mathtt{y} \qquad \mathtt{y}\,\mathtt{R}^*\,\mathtt{z}}{\mathtt{x}\,\mathtt{R}^*\,\mathtt{z}}.$$

Если рассматривать R как ориентированный граф, то R^* это отношение, соединяющее две вершины тогда и только тогда, когда существует путь, ведущий из одной вершины в другую.

1.2. Языки

1.2.1. Языки без переменных

Теперь, после того как введены индуктивные определения, их можно использовать для определения понятия *языка*. Вводимое понятие языка не учитывает второстепенные синтаксические детали, например, неважно, какую из форм записи применять: `3 + 4`, `+(3, 4)` или `3 4 +`. Данный терм будет представлен в абстрактной форме деревом. Каждый узел в дереве будет помечен символом. Количество дочерних вершин узла зависит от пометки узла: например, два дочерних узла, если пометка `+`, и нуль, если пометка 3 или 4.



Язык представляет собой набор символов, каждый их которых снабжён числовой характеристикой — *арностью* или, проще говоря, *количеством аргументов* символа. Символы без аргументов называются *константами*.

Множество термов языка есть множество деревьев, индуктивно определённых следующим правилом:

> если `f` это символ с `n` аргументами, $t_1, \ldots, t_n$ — термы, то $f(t_1, \ldots, t_n)$ — то есть дерево с корнем, помеченным `f`, и поддеревьями $t_1, \ldots, t_n$ — является термом.

1.2.2. Переменные

Представим, что перед нами стоит задача спроектировать язык для определения функций. Одна из возможностей — использовать константы `sin`, `cos`, `...` и символ $\circ$ с двумя аргументами. В таком языке возможно построить, к примеру, терм $\texttt{sin} \circ (\texttt{cos} \circ \texttt{sin})$.

Однако известно, что для определения функций проще использовать понятие, введённое Ф. Виетом (1540–1603), — понятие переменной. Функция, определённая выше, с помощью переменной может быть записана следующим образом: `sin(cos(sin x))`.

С 1930-х годов применяются и другие способы записи указанной выше функции: `x` $\mapsto$ `sin(cos(sin x))` или λ`x. sin(cos(sin x))`, где символы $\mapsto$ и λ используются для *связывания* переменной `x`. Явно указывая, какие переменные связаны, можно отличить аргументы функции от потенциальных параметров, а также зафиксировать порядок аргументов.

Символ $\mapsto$ был введён Н. Бурбаки около 1930, а символ λ — А. Чёрчем примерно в то же время. Нотация с λ является упрощённой версией предыдущей версии $\hat{\texttt{x}}$ `sin(cos(sin x))`, использовавшейся А. Н. Уайтхедом и Б. Расселом с 1900-х.

Определение `f = x` $\mapsto$ `sin(cos(sin x))` иногда записывается в форме `f x = sin(cos(sin x))`. Преимущество записи `f = x` $\mapsto$ `sin(cos(sin x))` в том, что здесь можно разделить две операции: конструирование функции `x` $\mapsto$ `sin(cos(sin x))` и само определение, которое даёт имя только что сконструированному объекту. В информатике часто бывает важным создавать объекты, не присваивая им имён.

В этой книге используется нотация, при которой рассматриваемая функция выглядит так: **fun** `x` $\to$ `sin(cos(sin x))`.

Терм **fun** `x` $\to$ `sin(cos(sin x))` задаёт функцию. Однако его подтерм `sin x` не задаёт ничего: это ни вещественное число, ни функция, так как содержит *свободную переменную*, значение которой неизвестно.

Для связывания переменных в термах необходимо расширить понятие терма, включив в него свободные переменные, которые будут связываться позднее. Это также потребует новых символов, подобных **fun**, которые выполняют связывание переменных в своих аргументах. Примерами таких символов связывания могут служить $\{\mid\}$, ∂/∂, $\int \mathtt{d}$, $\sum$, $\prod$, $\forall$, $\exists$ и т. д. В этой книге используется несколько подобных символов: уже упомянутый выше **fun**, а также символы **fix**, **let**, **fixfun** и др.

Арность символа `f` более не будет просто числом, вместо этого используется конечная последовательность чисел $(\mathtt{k}_1, \ldots, \mathtt{k}_n)$, которая показывает, что `f` связывает $\mathtt{k}_1$ переменных своего первого аргумента, $\mathtt{k}_2$ переменных второго аргумента, …, $\mathtt{k}_n$ переменных своего `n`-го аргумента.

Далее, после того как задан язык, то есть множество символов с предписанными арностями, и бесконечное множество переменных, можно индуктивно определить множество термов следующим образом:

— переменные являются термами;

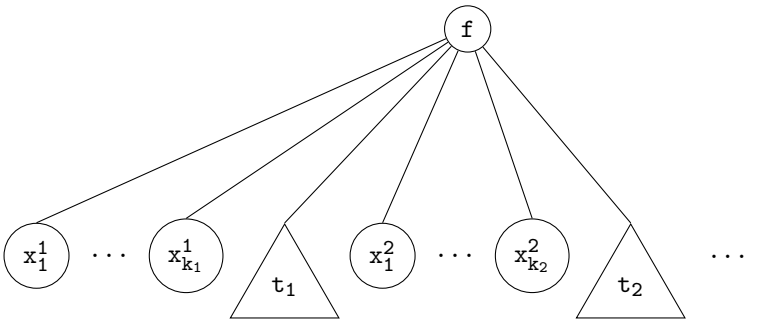
— если
  – $\mathtt{f}$ это символ арности $(\mathtt{k}_1, \ldots, \mathtt{k}_n)$,
  – $\mathtt{t}_1, \ldots, \mathtt{t}_n$ — термы,
  – $\mathtt{x}^1_1, \ldots, \mathtt{x}^1_{\mathtt{k}_1}, \ldots, \mathtt{x}^n_1, \ldots, \mathtt{x}^n_{\mathtt{k}_n}$ — переменные,

  то

$$\mathtt{f}(\mathtt{x}^1_1 \ldots \mathtt{x}^1_{\mathtt{k}_1}\, \mathtt{t}_1, \ldots, \mathtt{x}^n_1 \ldots \mathtt{x}^n_{\mathtt{k}_n}\, \mathtt{t}_n)$$

  это терм.

Запись $\mathtt{f}(\mathtt{x}^1_1 \ldots \mathtt{x}^1_{\mathtt{k}_1}\, \mathtt{t}_1, \ldots, \mathtt{x}^n_1 \ldots \mathtt{x}^n_{\mathtt{k}_n}\, \mathtt{t}_n)$ соответствует дереву:



Поясним данное определение на примере. Построим язык, в котором термы задают вещественные числа и функции на них, и который включает: две константы `sin` и `cos`, представляющие функции синуса и косинуса; символ α, называемый *применением*, такой что $\alpha(\mathtt{f}, \mathtt{x})$ это объект, получаемый применением функции `f` к объекту `x`; символ **fun** для построения новых функций. Таким образом, этот язык включает четыре символа: константы `sin` и `cos`, символ α арности `(0, 0)` и **fun** арности `(1)`; множество термов индуктивно определено следующим образом:

— переменные являются термами,
— `sin` является термом,
— `cos` является термом,
— если `t` и `u` это термы, то $\alpha(\mathtt{t}, \mathtt{u})$ является термом,
— если `t` это терм, а `x` — переменная, то `fun(x t)` является термом.

Введём упрощённую форму записи, в которой вместо терма $\alpha(\mathtt{t}, \mathtt{u})$ используется `t u`, а вместо `fun(x t)` — **fun** `x` $\to$ `t`.

К примеру, **fun** `x` $\to$ `sin(cos(sin x))` представляет собой терм данного языка.

1.2.3. Многосортные языки

В этой книге местами используются более общие, так называемые *многосортные языки*. Например, язык описания векторов с конечным набором константных символов, сложением и умножением на скаляр. В таком языке имеются два вида термов: термы, описывающие векторы, и термы, описывающие скаляры. В определении языка мы указываем, что символ + имеет два аргумента, оба являющихся векторами, а символ $\cdot$ имеет два аргумента, один из которых скаляр, а другой — вектор.

Для этого вводят двухэлементное множество *сортов* $\{\mathtt{vect}, \mathtt{scal}\}$, и символу $\cdot$ приписывают арность $(\mathtt{scal}, \mathtt{vect}, \mathtt{vect})$. Арность показывает, что в терме вида $\lambda \cdot \mathtt{v}$ терм λ должен иметь сорт scal, а терм v — сорт vect, сам же терм $\lambda \cdot \mathtt{v}$ имеет сорт vect.

Когда, кроме того, в языке присутствуют связанные переменные, арность символа f задаётся конечной последовательностью

$$((\mathtt{s}^1_1, \ldots, \mathtt{s}^1_{\mathtt{k}_1}, \mathtt{s}'^1), \ldots, (\mathtt{s}^{\mathtt{n}}_1, \ldots, \mathtt{s}^{\mathtt{n}}_{\mathtt{k}_\mathtt{n}}, \mathtt{s}'^{\mathtt{n}}), \mathtt{s}''),$$

указывающей, что символ имеет n аргументов, первый из которых принадлежит сорту $\mathtt{s}'^1$ и в котором связываются переменные сортов $\mathtt{s}^1_1, \ldots, \mathtt{s}^1_{\mathtt{k}_1}$, и т. д., результирующий терм принадлежит сорту $\mathtt{s}''$.

1.2.4. Свободные и связанные переменные

Множество переменных терма определяется структурной индукцией:

— $\mathtt{Var}(\mathtt{x}) = \{\mathtt{x}\}$,

— $\mathtt{Var}(\mathtt{f}(\mathtt{x}^1_1 \ldots \mathtt{x}^1_{\mathtt{k}_1}\, \mathtt{t}_1, \ldots, \mathtt{x}^{\mathtt{n}}_1 \ldots \mathtt{x}^{\mathtt{n}}_{\mathtt{k}_\mathtt{n}}\, \mathtt{t}_\mathtt{n})) =$
$\mathtt{Var}(\mathtt{t}_1) \cup \{\mathtt{x}^1_1, \ldots, \mathtt{x}^1_{\mathtt{k}_1}\} \cup \ldots \cup \mathtt{Var}(\mathtt{t}_\mathtt{n}) \cup \{\mathtt{x}^1_1, \ldots, \mathtt{x}^{\mathtt{n}}_{\mathtt{k}_\mathtt{n}}\}$.

Можно также определить множество свободных переменных терма:

— $\mathtt{FV}(\mathtt{x}) = \{\mathtt{x}\}$,

— $\mathtt{FV}(\mathtt{f}(\mathtt{x}^1_1 \ldots \mathtt{x}^1_{\mathtt{k}_1}\, \mathtt{t}_1, \ldots, \mathtt{x}^{\mathtt{n}}_1 \ldots \mathtt{x}^{\mathtt{n}}_{\mathtt{k}_\mathtt{n}}\, \mathtt{t}_\mathtt{n})) =$
$(\mathtt{FV}(\mathtt{t}_1) \setminus \{\mathtt{x}^1_1, \ldots, \mathtt{x}^1_{\mathtt{k}_1}\}) \cup \ldots \cup (\mathtt{FV}(\mathtt{t}_\mathtt{n}) \setminus \{\mathtt{x}^{\mathtt{n}}_1, \ldots, \mathtt{x}^{\mathtt{n}}_{\mathtt{k}_\mathtt{n}}\})$.

К примеру,

$\mathtt{Var}(\textbf{fun}\ \mathtt{x} \to \mathtt{sin}(\mathtt{cos}(\mathtt{sin}\,\mathtt{x}))) = \{\mathtt{x}\}$,

$\mathtt{FV}(\textbf{fun}\ \mathtt{x} \to \mathtt{sin}(\mathtt{cos}(\mathtt{sin}\,\mathtt{x}))) = \varnothing$.

Терм без свободных переменных называется *замкнутым*.

Высоту терма также можно определить посредством структурной индукции:

— $\texttt{Height(x)} = 0$,

— $\texttt{Height(f(}x_1^1 \ldots x_{k_1}^1\, t_1, \ldots, x_1^n \ldots x_{k_n}^n\, t_n\texttt{))} = 1 + \max(\texttt{Height}(t_1), \ldots, \texttt{Height}(t_n))$.

1.2.5. Подстановка

Первая операция, которую необходимо определить, это подстановка: действительно, назначение переменных не только быть связанными, но и использоваться для подстановки некоторых значений. Например, применение функции **fun** $\texttt{x} \to \texttt{sin(cos(sin x))}$ к терму $\texttt{2*}\pi$ подразумевает, что терм $\texttt{2*}\pi$ рано или поздно будет подставлен вместо переменной $\texttt{x}$ в терме $\texttt{sin(cos(sin x))}$.

Подстановка это отображение из переменных в термы с конечной областью определения. Другими словами, подстановка это конечное множество пар, в каждой из которых первый элемент это переменная, а второй — терм, причём такое множество, что каждая переменная входит в качестве первого элемента пар не более одного раза. Можно также определить подстановку как ассоциативный массив $\theta = t_1/x_1 \ldots t_n/x_n$.

Когда подстановка применяется к терму, каждое вхождение переменной $x_1, \ldots, x_n$ в терм заменяется на $t_1, \ldots, t_n$ соответственно.

Разумеется, замена осуществляется только на свободных переменных. К примеру, если выполняется подстановка терма 2 вместо переменной $\texttt{x}$ в терме $\texttt{x + 3}$, результатом будет $\texttt{2 + 3}$. Однако если 2 подставляется вместо $\texttt{x}$ в терме **fun** $\texttt{x} \to \texttt{x}$, который представляет тождественную функцию, результатом должно быть **fun** $\texttt{x} \to \texttt{x}$, а не **fun** $\texttt{x} \to \texttt{2}$.

Первая попытка строго определить применение подстановки к терму выглядит следующим образом:

— $\langle\theta\rangle x_i = t_i$,

— $\langle\theta\rangle\texttt{x} = \texttt{x}$, если $\texttt{x}$ не принадлежит области определения θ,

— $\langle\theta\rangle\texttt{f(}y_1^1 \ldots y_{k_1}^1\, u_1, \ldots, y_1^n \ldots y_{k_n}^n\, u_n\texttt{)} = \texttt{f(}y_1^1 \ldots y_{k_1}^1 \langle\theta|_{V\setminus\{y_1^1,\ldots,y_{k_1}^1\}}\rangle u_1, \ldots, y_1^n \ldots y_{k_n}^n \langle\theta|_{V\setminus\{y_1^n,\ldots,y_{k_n}^n\}}\rangle u_n\texttt{)}$,

где запись $\theta|_{V\setminus\{y_1,\ldots,y_k\}}$ используется для ограничения подстановки θ на множество $V \setminus \{y_1, \ldots, y_k\}$, то есть для исключения из подстановки всех пар, первый элемент которых принадлежит множеству $\{y_1, \ldots, y_k\}$.

Это определение имеет один недостаток: подстановки могут *захватывать переменные*. Например, терм **fun** $\mathtt{x} \to (\mathtt{x} + \mathtt{y})$ представляет функцию, которая прибавляет $\mathtt{y}$ к своему аргументу. Если заменить $\mathtt{y}$ на 4, получится терм, представляющий функцию, которая добавляет 4 к своему аргументу. Если заменить $\mathtt{y}$ на $\mathtt{z}$, получится терм **fun** $\mathtt{x} \to (\mathtt{x} + \mathtt{z})$, представляющий функцию, которая добавляет $\mathtt{z}$ к своему аргументу. Однако если заменить $\mathtt{y}$ на $\mathtt{x}$, получится функция **fun** $\mathtt{x} \to (\mathtt{x} + \mathtt{x})$, которая удваивает свой аргумент, вместо функции, которая добавляет $\mathtt{x}$ к своему аргументу, как следовало бы ожидать. Можно избежать этой проблемы, если поменять имя связанной переменной: связанные переменные это просто заглушки, их имена не играют существенной роли. Другими словами, в терме **fun** $\mathtt{x} \to (\mathtt{x} + \mathtt{y})$ можно заменить связанную переменную $\mathtt{x}$ на любую другую, за исключением, конечно, $\mathtt{y}$. Аналогично, подставляя в терм $\mathtt{u}$ термы $\mathtt{t}_1, \ldots, \mathtt{t}_n$ вместо переменных $\mathtt{x}_1, \ldots, \mathtt{x}_n$, можно поменять имена связанных переменных в $\mathtt{u}$, чтобы быть уверенным в том, что их имён нет среди $\mathtt{x}_1, \ldots, \mathtt{x}_n$, или среди свободных переменных $\mathtt{t}_1, \ldots, \mathtt{t}_n$, или среди свободных переменных $\mathtt{u}$, во избежание захвата.

Начнём с определения отношения эквивалентности на термах индукцией по высоте терма. Оно называется алфавитной эквивалентностью, или α-эквивалентностью, и соответствует переименованию переменных.

— $\mathtt{x} \sim \mathtt{x}$,

— $\mathtt{f}(\mathtt{y}^1_1 \ldots \mathtt{y}^1_{k_1} \mathtt{t}_1, \ldots, \mathtt{y}^n_1 \ldots \mathtt{y}^n_{k_n} \mathtt{t}_n) \sim \mathtt{f}(\mathtt{y'}^1_1 \ldots \mathtt{y'}^1_{k_1} \mathtt{t'}_1, \ldots, \mathtt{y'}^n_1 \ldots \mathtt{y'}^n_{k_n} \mathtt{t'}_n)$, если для всех $\mathtt{i}$ и для каждой последовательности свежих переменных $\mathtt{z}_1, \ldots, \mathtt{z}_{k_i}$ (то есть для переменных, не входящих свободно в $\mathtt{t}_i$, $\mathtt{t'}_i$) имеет место $\langle \mathtt{z}_1/\mathtt{y}^i_1, \ldots, \mathtt{z}_{k_i}/\mathtt{y}^i_{k_i} \rangle \mathtt{t}_i \sim \langle \mathtt{z}_1/\mathtt{y'}^i_1, \ldots, \mathtt{z}_{k_i}/\mathtt{y'}^i_{k_i} \rangle \mathtt{t'}_i$.

Например, термы **fun** $\mathtt{x} \to \mathtt{x} + \mathtt{z}$ и **fun** $\mathtt{y} \to \mathtt{y} + \mathtt{z}$ α-эквивалентны.

До конца книги мы работаем с термами *по модулю α-эквивалентности*, то есть подразумевая вместо термов классы α-эквивалентности термов.

Теперь можно определить операцию подстановки индукцией по высоте терма:

— $\theta \mathtt{x}_i = \mathtt{t}_i$,

— $\theta \mathtt{x} = \mathtt{x}$, если $\mathtt{x}$ не принадлежит области определения θ,

— $\theta \mathtt{f}(\mathtt{y}^1_1 \ldots \mathtt{y}^1_{k_1} \mathtt{u}_1, \ldots, \mathtt{y}^n_1 \ldots \mathtt{y}^n_{k_n} \mathtt{u}_n) = \mathtt{f}(\mathtt{z}^1_1 \ldots \mathtt{z}^1_{k_1} \theta \langle \mathtt{z}^1_1/\mathtt{y}^1_1, \ldots, \mathtt{z}^1_{k_1}/\mathtt{y}^1_{k_1} \rangle \mathtt{u}_1, \ldots, \mathtt{z}^n_1 \ldots \mathtt{z}^n_{k_n} \theta \langle \mathtt{z}^n_1/\mathtt{y}^n_1, \ldots, \mathtt{z}^n_{k_n}/\mathtt{y}^n_{k_n} \rangle \mathtt{u}_n)$, где $\mathtt{z}^1_1, \ldots, \mathtt{z}^1_{k_1}, \ldots, \mathtt{z}^n_1 \ldots \mathtt{z}^n_{k_n}$ это не входящие свободно в $\mathtt{f}(\mathtt{y}^1_1 \ldots \mathtt{y}^1_{k_1} \mathtt{u}_1, \ldots, \mathtt{y}^n_1 \ldots \mathtt{y}^n_{k_n} \mathtt{u}_n)$ и θ переменные.

К примеру, подстановка `2*x` вместо `y` в терме **fun** `x` $\to$ `x + y` даёт **fun** `z` $\to$ `z+(2*x)`. Выбор переменной `z` произволен, можно было бы взять `v` или `w`, и в результате получился бы тот же самый терм по модулю α-эквивалентности.

Композицией подстановок $\theta = \mathtt{t_1/x_1 \ldots t_n/x_n}$ и $\sigma = \mathtt{u_1/y_1 \ldots u_p/y_p}$ является подстановка

$$\theta \circ \sigma = \{\theta(\sigma \mathtt{z})/\mathtt{z} \mid \mathtt{z} \in \{\mathtt{x_1}, \ldots, \mathtt{x_n}, \mathtt{y_1}, \ldots, \mathtt{y_p}\}\}.$$

Можно доказать индукцией по высоте `t`, что для любого терма `t` имеет место

$$(\theta \circ \sigma)\mathtt{t} = \theta(\sigma \mathtt{t}).$$

1.3. Три способа задания семантики языка

Семантика языка программирования это бинарное отношение на множестве термов языка. Поскольку мы уже определили понятие языка и ввели инструменты для определения отношений, всё готово для описания трёх основных подходов к определению семантики. Семантика языка обычно задаётся функцией, индуктивным определением или рефлексивно-транзитивным замыканием явно заданного отношения. Эти три подхода называются соответственно *денотационной семантикой*, *операционной семантикой с большим шагом* и *операционной семантикой с малым шагом*.

1.3.1. Денотационная семантика

Денотационная семантика полезна для детерминированных языков. В этом случае для каждой программы `p` отношение входных и выходных данных, определяемое программой, является функцией, обозначаемой $[\![\mathtt{p}]\!]$. Отношение $\hookrightarrow$ определено следующим условием:

`p, e` $\hookrightarrow$ `s` тогда и только тогда, когда $[\![\mathtt{p}]\!]\mathtt{e} = \mathtt{s}$.

Конечно, это просто откладывает решение задачи до момента определения функции $[\![\mathtt{p}]\!]$. Для этого будут использованы два средства: явные определения функций и теорема о неподвижной точке. Подробнее об этом позднее.

1.3.2. Операционная семантика с большим шагом

Операционная семантика с большим шагом также называется *структурной операционной семантикой* (С.О.С.) или *естественной семантикой*. Она даёт индуктивное определение отношения $\hookrightarrow$.

1.3.3. Операционная семантика с малым шагом

Операционная семантика с малым шагом также называется *редукционной семантикой*. Она определяет отношение $\hookrightarrow$ в терминах другого отношения $\triangleright$, которое описывает элементарные шаги для последовательного преобразования исходного терма `t` к результирующему терму `s`.

Например, если запустить программу **fun** `x` $\to$ `(x * x) + x` со входом `4`, будет получен результат `20`. Но терм `(`**fun** `x` $\to$ `(x * x) + x) 4` не превращается в `20` за один шаг, сперва он преобразуется к `(4 * 4) + 4`, затем к `16 + 4` и только потом к `20`.

Наиболее важно не то отношение, которое связывает `(`**fun** `x` $\to$ `(x * x) + x) 4` и `20`, а $\triangleright$, которое связывает `(`**fun** `x` $\to$ `(x * x) + x) 4` и `(4 * 4) + 4`, а затем — терм `(4 * 4) + 4` с `16 + 4` и, наконец, терм `16 + 4` с `20`.

Как только определено $\triangleright$, отношение $\hookrightarrow$ может быть получено с помощью рефлексивно-транзитивного замыкания $\triangleright^*$ отношения $\triangleright$:

`t` $\hookrightarrow$ `s` тогда и только тогда, когда `t` $\triangleright^*$ `s` и `s` далее не редуцируем.

Тот факт, что `s` нередуцируем, означает, что внутри `s` больше нечего вычислять. Например, терм `20` нередуцируем, а терм `16 + 4` — нет. Терм `s` нередуцируем, если не существует терма $\mathtt{s}'$, такого что `s` $\triangleright$ $\mathtt{s}'$.

1.3.4. Незавершающиеся вычисления

Исполнение программы может выдавать результат, приводить к ошибке или вообще не завершаться. Ошибки можно рассматривать как особый вид результатов. Для незавершающихся программ существует несколько способов определения их семантики. Первая возможность состоит в том, чтобы считать, что не существует пары `(t, s)` в отношении $\hookrightarrow$, если `t` не завершается. Другая альтернатива предполагает добавление специального элемента $\bot$ (читается: «дно») во множество возможных выходных значений и пар `(t,` $\bot$`)` в отношение $\hookrightarrow$ для всех термов `t`, которые не завершаются.

Различие между данными подходами может казаться незначительным: не так уж сложно удалить все пары вида `(t,` $\bot$`)` или добавить таковые, если в отношении нет ни одной пары вида `(t, s)`. Однако читатели, знакомые с проблемами теории алгоритмов, заметят, что при добавлении пар `(t,` $\bot$`)` отношение $\hookrightarrow$ перестаёт быть рекурсивно перечислимым.

Глава 2. Язык PCF

В этой главе мы продемонстрируем несколько стилей задания семантики языка программирования, пользуясь одним примером: языком PCF — *языком программирования вычислимых функций* (Programming language for computable functions), называемым также Mini-ML.

2.1. Функциональный язык PCF

2.1.1. Программы как функции

В предыдущей главе мы наблюдали за тем, как детерминированная программа вычисляет значение функции, и вывели из этого наблюдения принципы денотационной семантики. Тот же процесс вычисления является основой целого класса языков программирования — функциональных языков, таких как Caml, Haskell или Lisp, которые традиционно используются в начале изучения теории языков программирования.

Цель этих языков заключается в сближении понятий программы и математической функции. Другими словами, идея в том, чтобы приблизить программы к их денотационной семантике.

Основными конструкциями языка PCF будут *явное построение* функции, записываемое как **fun** `x` $\to$ `t`, и *применение* функции к аргументу, обозначаемое `t u`.

Помимо этого, PCF содержит константы для каждого натурального числа (включая `0`), операции `+`, `-`, `*`, `/`, а также проверку на ноль **ifz** `t` **then** `u` **else** `v`. Сложение и умножение определены для всех натуральных чисел, чтобы сделать таковым вычитание, будем использовать соглашение: если `n` $<$ `m`, то `n - m = 0`. Деление является обычным целочисленным делением, деление на `0` приводит к ошибке.

2.1.2. Функции как объекты первого класса

Во многих языках программирования допускается определение функции, которая либо принимает другую функцию в качестве аргумента, либо возвращает её в качестве результата, однако зачастую это требует некоторого специального синтаксиса, отличающегося от синтаксиса для передачи

обычных аргументов, таких как целые числа или строки. В функциональных языках функции определяются единообразно, независимо от того, являются ли их аргументы числами или же функциями.

Например, композиция функции с самой собой определяется как **fun** `f` $\to$ **fun** `x` $\to$ `f(f x)`.

Чтобы обозначить тот факт, что функции не рассматриваются специальным образом, а значит, что их можно использовать как аргументы или результаты других функций, мы будем говорить, что функции являются *объектами первого класса*.

2.1.3. Функции с несколькими аргументами

В языке PCF нет символа для построения функций с несколькими аргументами. Эти функции определяются как функции с одним аргументом, используя изоморфизм `(A × B) → C = A → (B → C)`. Например, функция, ставящая в соответствие числам `x` и `y` выражение `x*x+y*y`, определяется как функция, ставящая в соответствие числу `x` функцию, которая в свою очередь ставит в соответствие числу `y` выражение `x*x+y*y`. Окончательно получаем: **fun** `x` $\to$ **fun** `y` $\to$ `x*x+y*y`.

Теперь, чтобы применить функцию `f` к параметрам `3` и `4` необходимо сначала применить её к `3`, получив терм `f 3`, который представляет функцию, ставящую в соответствие числу `y` выражение `3*3+y*y`, а затем к `4`, получив терм `(f 3) 4`. Так как по нашему соглашению применение ассоциируется слева направо, последний терм можно записать просто как `f 3 4`.

2.1.4. Без присваиваний

В отличие от таких языков, как Caml или Java, важной чертой PCF является полное отсутствие *присваиваний*. В нём нет конструкций вида `x := t` или `x = t` для обозначения присваивания «переменной» нового значения. В главе 7, однако, мы представим расширение языка PCF с присваиваниями.

2.1.5. Рекурсивные определения

Далеко не все математические функции могут быть явно определены. В школьных учебниках, к примеру, функцию возведения в степень обычно определяют как

$$\mathtt{x,n} \mapsto \underbrace{\mathtt{x} \times \cdots \times \mathtt{x}}_{\mathtt{n}\text{ раз}}$$

или с помощью определения по индукции.

В языках программирования используются похожие конструкции: циклы и рекурсивные определения. PCF включает специальную конструкцию для определения рекурсивных функций.

Часто говорится, что функция является рекурсивной, если она используется в собственном определении. Это, однако, бессмыслица: в языках программирования, как и вообще везде, при попытке использования подобных определений мы попадаем в порочный круг. Нельзя «определить» функцию `fact` как **fun** `n` → **ifz** `n` **then** `1` **else** `n*(fact(n-1))`. И вообще, мы не можем определить функцию `f` термом `G`, содержащим вхождение `f`. Однако мы можем определить функцию `f` как неподвижную точку функции **fun** `f` → `G`. Например, функцию `fact` можно определить как неподвижную точку функции **fun** `f` → **fun** `n` → **ifz** `n` **then** `1` **else** `n*(f(n-1))`.

Имеет ли эта функция неподвижную точку? И если имеет, является ли эта неподвижная точка единственной? А если она не единственная, то какую из них мы получаем? Оставим пока эти вопросы без ответа, считая просто, что рекурсивная функция определяется как неподвижная точка.

В языке PCF символ **fix** связывает переменную в своём аргументе, а терм **fix** `f G` используется для обозначения неподвижной точки функции **fun** `f` → `G`. Функция `fact` теперь может быть определена как **fix** `f` **fun** `n` → **ifz** `n` **then** `1` **else** `n*(f(n-1))`.

Заметим снова, что использование символа **fix** позволяет нам построить функцию вычисления факториала без необходимости явно указывать её имя.

2.1.6. Определения

Теоретически можно вообще отказаться от определений, заменяя всюду определяемые символы на их определения. Однако с использованием определений программы становятся проще и понятнее.

Добавим в язык PCF последнюю конструкцию: **let** `x = t` **in** `u`. Вхождения переменной `x` в `u` являются связанными, а её же вхождения в `t` нет. Символ **let** является бинарным оператором, связывающим переменную во втором аргументе.

2.1.7. Язык PCF

Язык PCF содержит:

— символ **fun** с одним аргументом, который связывает переменную в своём аргументе;

— символ α с двумя аргументами, который не связывает никакие переменные (применение α(`t`, `u`), упрощённо записываемое как `t u`);

— бесконечное число констант, представляющих натуральные числа;

— четыре символа `+`, `-`, `*`, `/` с двумя аргументами каждый, не связывающие никакие переменные;

— символ **ifz** с тремя аргументами, который не связывает никакие переменные;

— символ **fix** с одним аргументом, который связывает переменную в своём аргументе;

— символ **let** с двумя аргументами, который связывает переменную во втором аргументе.

Другими словами, синтаксис PCF определяется индуктивно следующим образом:

`t = x`

| **fun** `x` → `t`

| `t t`

| `n`

| `t + t` | `t - t` | `t * t` | `t/t`

| **ifz** `t` **then** `t` **else** `t`

| **fix** `x t`

| **let** `x = t` **in** `t`

Несмотря на столь малый размер, язык PCF является полным по Тьюрингу, то есть на нём можно запрограммировать любую вычислимую функцию.

Упражнение 2.1. Напишите программу на языке PCF, которая принимает на вход два натуральных числа `n` и `p` и возвращает $\mathtt{n}^{\mathtt{p}}$.

Упражнение 2.2. Напишите программу на языке PCF, которая принимает на вход натуральное число `n` и возвращает `1`, если число `n` является простым, и `0` в противном случае.

Упражнение 2.3 (многочлены в PCF)**.**

(1) Напишите программу на языке PCF, которая принимает на вход натуральное число `q` и возвращает наибольшее натуральное число `u`, такое что

$$\texttt{u(u+1)/2} \leqslant \texttt{q}.$$

(2) Функцией Кантора `K` называется функция, действующая из $\mathbb{N}^2$ в $\mathbb{N}$, определяемая как

$$\textbf{fun}\ \texttt{n} \to \textbf{fun}\ \texttt{p} \to \texttt{(n+p)(n+p+1)/2+n}.$$

Пусть функция $\texttt{K}'$ действует из $\mathbb{N}$ в $\mathbb{N}^2$ и определяется как

$$\textbf{fun}\ \texttt{q} \to \texttt{(q - (u(u+1)/2), u - q + u(u+1)/2),}$$

где `u` — наибольшее натуральное число, такое что $\texttt{u(u+1)/2} \leqslant \texttt{q}$.

Пусть `n` и `p` натуральные числа. Покажите, что наибольшее натуральное число `u`, такое что $\texttt{u(u+1)/2} \leqslant \texttt{(n+p)(n+p+1)/2+n}$ равно `n+p`. Затем покажите, что $\texttt{K} \circ \texttt{K}' = \texttt{id}$. Пользуясь этим фактом, покажите, что `K` является биективным отображением из $\mathbb{N}^2$ на $\mathbb{N}$.

(3) Пусть `L` — функция **fun** `n` $\to$ **fun** `p` $\to$ `(K n p)+1`. Многочлен с целыми коэффициентами $\texttt{a}_0+\texttt{a}_1\texttt{X}+\cdots+\texttt{a}_\texttt{i}\texttt{X}^\texttt{i}+\cdots+\texttt{a}_\texttt{n}\texttt{X}^\texttt{n}$ может быть представлен целым числом $\texttt{L a}_0\ \texttt{(L a}_1\ \texttt{(L a}_2 \ldots \texttt{(L a}_\texttt{n}\ \texttt{0)}\ldots\texttt{))}$.

Напишите программу на языке PCF, которая принимает на вход два натуральных числа и возвращает значение многочлена, заданного первым числом, в точке, заданной вторым.

2.2. Операционная семантика с малым шагом

2.2.1. Правила

Применим программу **fun** `x` $\to$ `2*x` к константе `3`. Получится терм (**fun** `x` $\to$ `2*x`) `3`. Теперь, согласно принципам операционной семантики с малым шагом, будем вычислять значение этого терма шаг за шагом с тем, чтобы, если повезёт, получить в результате `6`. Первым шагом в процессе упрощения терма является *передача параметра*, то есть замена формального параметра `x` на фактический аргумент `3`. После него исходный терм превращается в `2*3`. На втором шаге вычислений терм `2*3` даёт результат `6`. Первый малый шаг, передача параметра, может выполняться всякий

раз, когда терм имеет вид `(`**fun** `x` $\to$ `t)u`, то есть функция **fun** `x` $\to$ `t` применяется к аргументу `u`. Как следствие, мы определили правило, называемое правилом *β-редукции*:

`(`**fun** `x` $\to$ `t) u` $\longrightarrow$ `(u/x)t.`

Отношение `t` $\longrightarrow$ `u` читается как «`t` редуцируется (или сводится) к `u`». Второй из упомянутых ранее шагов можно обобщить так:

`p` $\otimes$ `q` $\longrightarrow$ `n` (если `p` $\otimes$ `q = n`),

где $\otimes$ является одной из четырёх определённых выше операций, включённых в язык PCF. Добавим аналогичные правила для условий:

ifz `0` **then** `t` **else** `u` $\longrightarrow$ `t`,

ifz `n` **then** `t` **else** `u` $\longrightarrow$ `u` (если `n` — число, отличное от `0`);

правило для неподвижных точек:

fix `x t` $\longrightarrow$ `(`**fix** `x t/x)t`;

и правило для **let**:

let `x = t` **in** `u` $\longrightarrow$ `(t/x)u.`

Редексом назовём терм, который можно редуцировать. Другими словами, терм `t` является редексом, если существует такой терм `u`, что `t` $\longrightarrow$ `u`.

2.2.2. Числа

Читатель может совершенно справедливо заметить, что правило

`p` $\otimes$ `q` $\longrightarrow$ `n (если p` $\otimes$ `q = n),`

примером применения которого является преобразование `2*3` $\longrightarrow$ `6`, на самом деле не объясняет семантику арифметических операций, оно лишь заменяет соответствующую операцию PCF на математическую. Этот выбор, однако, объясняется тем фактом, что мы нисколько не интересуемся семантикой арифметических операций, вместо этого нашей целью является исследование семантики других языковых конструкций.

Для определения семантики арифметических операций в PCF без обращения к их математическому смыслу следовало бы рассматривать версию языка PCF без числовых констант, в которой имеется только одна

константа для числа `0` и символ `S` — «последователь» (successor) — с одним аргументом. Число `3`, к примеру, представимо в таком случае в виде терма `S(S(S(0)))`. Затем можно добавить правила малых шагов:

$$\texttt{0 + u} \longrightarrow \texttt{u}$$

$$\texttt{S(t) + u} \longrightarrow \texttt{S(t + u)}$$

$$\texttt{0 - u} \longrightarrow \texttt{0}$$

$$\texttt{t - 0} \longrightarrow \texttt{t}$$

$$\texttt{S(t) - S(u)} \longrightarrow \texttt{t - u}$$

$$\texttt{0 * u} \longrightarrow \texttt{0}$$

$$\texttt{S(t) * u} \longrightarrow \texttt{t * u + u}$$

$$\texttt{t/S(u)} \longrightarrow \textbf{ifz } \texttt{t - u} \textbf{ then } \texttt{0} \textbf{ else } \texttt{S((t - S(u))/S(u))}$$

Заметим, что для большей строгости следовало бы добавить правило деления на `0`, которое возбуждало бы исключение: `error`.

Упражнение 2.4 (нумералы Чёрча)**.** Вместо введения символов `0` и `S` и представления числа `n` термом `S(S(...(0)...))` мы можем представлять число `n` посредством терма $\textbf{fun z} \to \textbf{fun s} \to \texttt{s(s(...(s z)...))}$. Покажите, что с помощью этого представления можно запрограммировать сложение и умножение. Покажите также, что можно запрограммировать функцию, проверяющую, является ли заданный терм представлением числа `0`.

Упражнение 2.5 (позиционные нумералы)**.** Можно сказать, что представления чисел как с помощью символов `0` и `S`, так и через нумералы Чёрча, является неэффективным, поскольку размер терма, представляющего число, растёт линейно относительно этого числа — как представление в унарной записи, где для выражения числа `n` необходимо `n` символов, — а не логарифмически, что имеет место при традиционной позиционной записи. В качестве альтернативы можно взять символ `z` для представления числа `0` и две функции `O` и `I`, соответствующие функциям $\texttt{n} \mapsto \texttt{2*n}$ и $\texttt{n} \mapsto \texttt{2*n+1}$. Число `26` тогда можно было бы представить термом `O(I(O(I(I(z)))))`. Если обратить порядок применения функций и отбросить скобки, то получим `IIOIO` — представление числа в двоичной системе счисления.

Напишите правила операционной семантики с малым шагом для арифметических операций в таком языке.

2.2.3. Эквивалентность (congruence)

Используя правила семантики с малым шагом, получаем:

$$(\textbf{fun}\ \mathtt{x} \to 2 * \mathtt{x})\ 3 \longrightarrow 2 * 3 \longrightarrow 6.$$

Таким образом, обозначив через $\longrightarrow^*$ рефлексивно-транзитивное замыкание отношения $\longrightarrow$, можно записать $(\textbf{fun}\ \mathtt{x} \to 2 * \mathtt{x})\ 3 \longrightarrow^* 6$.

Однако это определение отношения $\longrightarrow^*$ не позволяет редуцировать терм $(2 + 3) + 4$ к терму 9. Действительно, чтобы редуцировать терм вида $\mathtt{t} + \mathtt{u}$ термы $\mathtt{t}$ и $\mathtt{u}$ должны быть числовыми константами, тогда как в нашем примере терм $2 + 3$ является суммой, а не константой. Первым шагом следовало бы вычислить $2 + 3$, что даёт число 5. Тогда второй шаг редуцирует $5 + 4$ к 9. Таким образом, проблема заключается в том, что, согласно нашему определению, $2 + 3$ редуцируется к 5, но $(2 + 3) + 4$ не редуцируется к $5 + 4$.

Необходимо определить другое отношение, в котором правила можно применять к любому подтерму редуцируемого терма. Построим индуктивное определение отношения $\rhd$:

$$\frac{\mathtt{t} \rhd \mathtt{u}}{\mathtt{t\ v} \rhd \mathtt{u\ v}},$$

$$\frac{\mathtt{t} \rhd \mathtt{u}}{\mathtt{v\ t} \rhd \mathtt{v\ u}},$$

$$\frac{\mathtt{t} \rhd \mathtt{u}}{\textbf{fun}\ \mathtt{x} \to \mathtt{t} \rhd \textbf{fun}\ \mathtt{x} \to \mathtt{u}},$$

$$\frac{\mathtt{t} \rhd \mathtt{u}}{\mathtt{t} + \mathtt{v} \rhd \mathtt{u} + \mathtt{v}},$$

$\ldots$

Можно показать, что терм является редексом согласно отношению $\rhd$, если и только если один из его подтермов является редексом согласно отношению $\longrightarrow$.

2.2.4. Пример

В качестве примера применения правил семантики с малым шагом для языка PCF вычислим факториал 3.

$$(\textbf{fix}\ \mathtt{f}\ \textbf{fun}\ \mathtt{n} \to \textbf{ifz}\ \mathtt{n}\ \textbf{then}\ 1\ \textbf{else}\ \mathtt{n} * (\mathtt{f}\ (\mathtt{n} - 1)))\ 3$$

$\triangleright$ (**fun** n $\to$ **ifz** n **then** 1 **else**
n*((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1))) (n - 1))) 3

$\triangleright$ **ifz** 3 **then** 1 **else**
3*((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1))) (3 - 1))

$\triangleright$ 3*((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1))) (3 - 1))

$\triangleright$ 3*((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1))) 2)

$\triangleright$ 3*((**fun** n $\to$ **ifz** n **then** 1 **else**
n*((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1))) (n - 1))) 2)

$\triangleright$ 3*(**ifz** 2 **then** 1 **else**
2*((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1))) (2 - 1)))

$\triangleright$ 3*(2*((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1))) (2 - 1)))

$\triangleright$ 3*(2*((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1))) 1))

$\triangleright$ 3*(2*((**fun** n $\to$ **ifz** n **then** 1 **else** n*
((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1)))(n - 1))) 1))

$\triangleright$ 3*(2*(**ifz** 1 **then** 1 **else**
1*((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1))) (1 - 1))))

$\triangleright$ 3*(2*(1*
((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1))) (1 - 1))))

$\triangleright$ 3*(2*(1*((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1))) 0)))

$\triangleright$ 3*(2*(1*((**fun** n $\to$ **ifz** n **then** 1 **else**
n*((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1))) (n - 1))
) 0)))

$\triangleright$ 3*(2*(1*((**ifz** 0 **then** 1 **else**
0*((**fix** f **fun** n $\to$ **ifz** n **then** 1 **else** n*(f (n - 1)))(0 - 1))))))

$\triangleright$ 3*(2*(1*1)) $\triangleright$ 3*(2*1) $\triangleright$ 3*2 $\triangleright$ 6

2.2.5. Нередуцируемые замкнутые термы

Терм t называется нередуцируемым, если его нельзя редуцировать с помощью отношения $\triangleright$, то есть, если не существует такого терма u, что

$\mathtt{t} \rhd \mathtt{u}$.

Теперь можно определить отношение «терм u является результатом вычисления терма t», где t — замкнутый терм, следующим образом: $\mathtt{t} \hookrightarrow \mathtt{u}$, если и только если $\mathtt{t} \rhd^* \mathtt{u}$, причём u нередуцируем. В этом случае терм u должен быть замкнут. Наконец, отношение «программа p с входными данными $\mathtt{e}_1, \ldots, \mathtt{e}_n$ возвращает s» можно записать просто как $\mathtt{p}\ \mathtt{e}_1 \ldots \mathtt{e}_n \hookrightarrow \mathtt{s}$.

Упражнение 2.6 (классификация нередуцируемых замкнутых термов)**.** Покажите, что терм является нередуцируемым и замкнутым тогда и только тогда, когда он имеет одну из следующих форм:

— **fun** x $\to$ t, где t нередуцируем и не содержит свободных переменных за исключением, быть может, x;

— n, где n — число;

— $\mathtt{V}_1\mathtt{V}_2$, где $\mathtt{V}_1$ и $\mathtt{V}_2$ нередуцируемые замкнутые термы, причём $\mathtt{V}_1$ не имеет форму **fun** x $\to$ t;

— $\mathtt{V}_1 \otimes \mathtt{V}_2$, где $\mathtt{V}_1$ и $\mathtt{V}_2$ нередуцируемые замкнутые термы и не являются одновременно числовыми константами;

— **ifz** $\mathtt{V}_1$ **then** $\mathtt{V}_2$ **else** $\mathtt{V}_3$, где $\mathtt{V}_1$, $\mathtt{V}_2$ и $\mathtt{V}_3$ нередуцируемые замкнутые термы и $\mathtt{V}_1$ не является числом.

Числа и нередуцируемые замкнутые термы вида **fun** x $\to$ t называются *значениями*. Если результат вычисления является значением, мы связываем значение с исходным термом и говорим, что это значение является значением терма (терм вычисляется к этому значению).

К сожалению, значения не являются единственно возможными результатами. Например, терм (**fun** x $\to$ x) 1 2 можно редуцировать до терма 1 2, который нередуцируем и замкнут, то есть действительно является результатом вычисления терма (**fun** x $\to$ x) 1 2. Однако этот результат не имеет смысла, потому что мы не можем применить единицу, не являющуюся функцией, к двойке. Нередуцируемые замкнутые термы, не являющиеся значениями, называют *тупиковыми* (stuck). Они могут принимать одну из следующих форм: $\mathtt{V}_1\ \mathtt{V}_2$, где $\mathtt{V}_1$ и $\mathtt{V}_2$ нередуцируемые замкнутые термы, причём $\mathtt{V}_1$ не является функцией **fun** x $\to$ t (например, 1 2); $\mathtt{V}_1 \otimes \mathtt{V}_2$, где $\mathtt{V}_1$ и $\mathtt{V}_2$ нередуцируемы, замкнуты и хотя бы один из них не является числом (например, 1 + (**fun** x $\to$ x)); **ifz** $\mathtt{V}_1$ **then** $\mathtt{V}_2$ **else** $\mathtt{V}_3$, где $\mathtt{V}_1$, $\mathtt{V}_2$ и $\mathtt{V}_3$ нередуцируемы, замкнуты и $\mathtt{V}_1$ не является числом (например, **ifz** (**fun** x $\to$ x) **then** 1 **else** 2).

Упражнение 2.7. Каковы значения следующих PCF-термов согласно операционной семантике с малым шагом?

(1) `(`**fun** `x` $\to$ **fun** `x` $\to$ `x) 2 3`;

(2) `(`**fun** `x` $\to$ **fun** `y` $\to$ `((`**fun** `x` $\to$ `(x + y)) x)) 5 4`.

Упражнение 2.8 (статическое связывание)**.** Каково значение терма

let `x = 4` **in let** `f = `**fun** `y` $\to$ `y + x` **in let** `x = 5` **in** `f 6`

с точки зрения операционной семантики с малым шагом, 10 или 11? Первые версии языка Lisp давали значение 11 вместо 10. В таких случаях говорят, что имеет место *динамическое* связывание.

2.2.6. Незавершающиеся вычисления

Легко увидеть, что отношение $\hookrightarrow$ не является тотальным, то есть существуют термы `t`, для которых не существует такого терма `u`, что `t` $\hookrightarrow$ `u`. Например, терм `b = `**fix** `x x` редуцируется к себе и только к себе. Его невозможно свести ни к какому нередуцируемому терму.

Упражнение 2.9. Пусть $\mathtt{b}_1$ `= (`**fix** `f (`**fun** `x` $\to$ `(f x))) 0`. Укажите все термы, получаемые редукцией этого терма. Существует ли в этом случае результат вычислений?

Упражнение 2.10 (оператор неподвижной точки Карри)**.**

(1) Пусть `t` некоторый терм, `u = (`**fun** `y` $\to$ `(t (y y)))(`**fun** `y` $\to$ `(t (y y)))`. Покажите, что `u` редуцируется к `t u`.

(2) Пусть `t` некоторый терм, `v = (`**fun** `y` $\to$ `((`**fun** `x` $\to$ `t) (y y)))(`**fun** `y` $\to$ `((`**fun** `x` $\to$ `t) (y y)))`. Покажите, что `v` редуцируется к `(v/x)t`.

(3) Запишите терм `u`, эквивалентный терму `b = `**fix** `x x`, не используя символ **fix**. Опишите термы, которые можно получить с помощью редукции. Будут ли вычисления иметь в этом случае результат?

Таким образом, можно утверждать, что символ **fix** из PCF является в некотором смысле избыточным. Однако эта избыточность впоследствии, когда в PCF будут добавлены типы, исчезнет.

2.2.7. Слияние (confluence)

Возможно ли в процессе редукции получить из замкнутого терма несколько различных результатов? И вообще, можно ли свести терм к нескольким различным нередуцируемым термам? Ответ на эти вопросы отрицательный. Фактически каждая PCF-программа является детерминированной, но это не такое уж тривиальное свойство. Давайте выясним, почему.

Терм `(3+4)+(5+6)` имеет два подтерма, каждый из которых является редексом. Мы можем начать, редуцировав сначала `3 + 4` к `7` или `5 + 6` к `11`. Действительно, терм `(3 + 4) + (5 + 6)` редуцируется и к `7 + (5 + 6)`, и к `(3 + 4) + 11`. К счастью, оба эти терма можно редуцировать далее, и, продолжая вычисления, мы приходим к одному и тому же результату `18` в обоих случаях.

Чтобы доказать, что любой терм можно свести не более чем к одному нередуцируемому терму, нужно показать, что если два вычисления, стартовавшие на одном и том же терме, привели к двум различным термам, то они неминуемо рано или поздно достигнут одного и того же нередуцируемого терма.

Этот факт является следствием того, что отношение $\triangleright$ обладает свойством *слияния*. Говорят, что отношение `R` обладает свойством слияния, если всякий раз, когда имеет место $\texttt{a}\ \texttt{R}^*\ \texttt{b}_1$ и $\texttt{a}\ \texttt{R}^*\ \texttt{b}_2$, существует некоторое `c`, такое что $\texttt{b}_1\ \texttt{R}^*\ \texttt{c}$ и $\texttt{b}_2\ \texttt{R}^*\ \texttt{c}$.

Свойство слияния в действительности влечёт за собой то, что любой терм может быть сведён к не более чем одному нередуцируемому терму. Если терм `t` можно редуцировать к двум далее нередуцируемым термам $\texttt{u}_1$ и $\texttt{u}_2$, то $\texttt{t} \triangleright^* \texttt{u}_1$ и $\texttt{t} \triangleright^* \texttt{u}_2$. В силу свойства слияния отношения $\triangleright$ существует такой терм `v`, что $\texttt{u}_1 \triangleright^* \texttt{v}$ и $\texttt{u}_2 \triangleright^* \texttt{v}$. Так как $\texttt{u}_1$ нередуцируем, то единственный такой терм `v`, что $\texttt{u}_1 \triangleright^* \texttt{v}$, есть сам $\texttt{u}_1$. Следовательно, $\texttt{u}_1 = \texttt{v}$ и точно так же $\texttt{u}_2 = \texttt{v}$. Отсюда получаем, что $\texttt{u}_1 = \texttt{u}_2$. Другими словами, `t` сводится к не более чем одному нередуцируемому терму.

Мы не будем доказывать здесь свойство слияния отношения $\triangleright$. Идея заключается в том, что когда терм `t` содержит два редекса $\texttt{r}_1$ и $\texttt{r}_2$, терм $\texttt{t}_1$ получен редуцированием $\texttt{r}_1$, а терм $\texttt{t}_2$ получен редуцированием $\texttt{r}_2$, то в терме $\texttt{t}_1$ можно найти остатки редекса $\texttt{r}_2$ и редуцировать их. Похожим образом можно редуцировать остатки $\texttt{r}_1$ в $\texttt{t}_2$, получив тот же терм. Например, редуцируя `5 + 6` в `7 + (5 + 6)` и редуцируя `3 + 4` в `(3 + 4) + 11`, мы получаем один и тот же терм `7 + 11`.

2.3. Стратегии редукции

2.3.1. Понятие стратегии

Поскольку каждый PCF-терм может иметь не более одного результата (благодаря свойству, отмеченному выше), неважно, в каком порядке вычислять редексы, входящие в терм: если мы придём к нередуцируемому терму, то он всегда будет одним и тем же. Однако возможна ситуация, при которой одна последовательность редукций приводит к нередуцируемому терму, а другая нет. Пусть, для примера, `C` является термом вида **fun** `x` $\to$ `0`, а $\mathtt{b}_1$ вида (**fix** `x` (**fun** `x` $\to$ `(f x)`)) `0`. Терм $\mathtt{b}_1$ сводится к $\mathtt{b}_2$ = (**fun** `x` $\to$ (**fix** `f` (**fun** `x` $\to$ `(f x)`) `x`)) `0`, а затем снова к $\mathtt{b}_1$. Терм `C` $\mathtt{b}_1$ содержит несколько редексов и может быть сведён либо к `0`, либо к `C` $\mathtt{b}_2$, который в свою очередь также содержит несколько редексов и может быть приведён либо к `0`, либо к `C` $\mathtt{b}_1$ (а также и к другим термам). Редуцируя всякий раз самый внутренний редекс, можно получить бесконечную последовательность редукций `c` $\mathtt{b}_1$ $\rhd$ `C` $\mathtt{b}_2$ $\rhd$ `C` $\mathtt{b}_1$ $\rhd$ $\cdots$, тогда как редуцированием внешнего редекса всегда можно получить `0`.

Этот пример может показаться исключением, ведь он содержит функцию `C`, которая игнорирует свой аргумент; заметьте, однако, что условная конструкция `ifz` ведёт себя схожим образом. При вычислении факториала числа `3` можно наблюдать аналогичное поведение. В терме

ifz `0` **then** `1` **else** `0*((`**fix** `f` **fun** `n` $\to$ **ifz** `n` **then** `1` **else** `n*(f(n-1)))(0-1))`

имеется несколько редексов. При выборе внешнего редекса получаем `1` (остальные редексы при этом пропадают), тогда как редуцируя

fix `f` **fun** `n` $\to$ **ifz** `n` **then** `1` **else** `n*(f(n-1))`

получаем бесконечную последовательность редукций. Другими словами, терм `fact 3` можно редуцировать к `6`, но можно и получить вычисления, продолжающиеся вечно.

Оба терма `C` $\mathtt{b}_1$ и `fact 3` дают единственный результат, но не все последовательности редукций позволяют к нему прийти.

Поскольку терм `C` $\mathtt{b}_1$ имеет значение `0`, согласно семантике PCF *вычислитель*, то есть программа, которая принимает на вход PCF-терм и возвращает его значение, должна при вычислении `C` $\mathtt{b}_1$ вернуть `0`. Попробуем воспользоваться несколькими современными компиляторами для того, чтобы провести такие вычисления. В языке Caml программа

```
let rec f x = f x in let g x = 0 in g (f 0)
```

не завершается. Та же проблема и с программой на Java:

```
class Omega {
  static int f (int x) {return f(x);}
  static int g (int x) {return 0;}
  static public void main (String[] args) {
    System.out.println(g(f(0)));
  }
}
```

И только очень небольшое число компиляторов для языков, использующих *вызов по имени* или *ленивые* вычисления, таких как Haskell, Lazy-ML или Gaml, позволяют получить завершающуюся программу, которая сможет вычислить результат этого терма.

Причина проблем в том, что семантика PCF с малым шагом не соответствует семантике Java или Caml. Фактически она является слишком общей, при наличии нескольких редексов она не указывает, какой именно редекс должен быть редуцирован. По умолчанию она рассчитывает на завершение всех программ, результат которых в принципе может быть вычислен. В определении этой семантики отсутствует важный компонент: понятие стратегии, определяющей порядок выбора редексов.

Стратегией называется частичная функция, которая каждому терму из своей области определения ставит в соответствие один из его редексов. Имея стратегию $\mathtt{s}$, можно определить другую семантику, заменив отношение $\triangleright$ на новое отношение $\triangleright_{\mathtt{s}}$, определяемое следующим: $\mathtt{t} \triangleright_{\mathtt{s}} \mathtt{u}$, если определён $\mathtt{s\ t}$ и терм $\mathtt{u}$ получен вычислением редекса $\mathtt{s\ t}$ в $\mathtt{t}$. Теперь можно определить отношение $\triangleright_{\mathtt{s}}^{*}$ как рефлексивно-транзитивное замыкание отношения $\triangleright_{\mathtt{s}}$, а также отношение $\hookrightarrow_{\mathtt{s}}$ указанным выше способом.

Альтернативой определению стратегии может быть такое ослабление правил редукции, в особенности правила эквивалентности, чтобы можно было вычислять только некоторые специальные редексы.

2.3.2. Слабая редукция

Перед определением различных стратегий вычисления терма $\mathtt{C\ b_1}$ рассмотрим ещё один пример, показывающий слишком большую свободу определённой выше операционной семантики и мотивирующий введение стратегий или ослабление правил редукции. Применим программу $\mathbf{fun}\ \mathtt{x} \to \mathtt{x + (4 + 5)}$ к константе $\mathtt{3}$. Получим терм $(\mathbf{fun}\ \mathtt{x} \to \mathtt{x + (4 + 5)})\ \mathtt{3}$, содержащий два редекса. Возможны два варианта редукции: $\mathtt{3 + (4 + 5)}$ и $(\mathbf{fun}\ \mathtt{x} \to \mathtt{x + 9})\ \mathtt{3}$. Первый вариант соответствует исполнению программы, а второй нет. Обычно говорят, что при вычислении тела функции до

передачи ей аргументов происходит *оптимизация* или *специализация* программы.

Слабая стратегия редукции никогда не вычисляет редекс, находящийся внутри **fun**. Таким образом, слабая редукция не специализирует программы, она их лишь исполняет. Отсюда следует, что при слабой стратегии все термы вида **fun** x → t являются нередуцируемыми.

Другой способ определения слабой редукции заключается в ослаблении правил редукции или, точнее, в удалении правила

$$\frac{\mathtt{t} \rhd \mathtt{u}}{\textbf{fun}\ \mathtt{x} \to \mathtt{t} \rhd \textbf{fun}\ \mathtt{x} \to \mathtt{u}}.$$

Упражнение 2.11 (классификация слабо нередуцируемых замкнутых термов)**.**

(1) Покажите, что при условии слабой редукции замкнутый нередуцируемый терм должен иметь одну из следующих форм:
   - **fun** x → t, где t не содержит свободных переменных за исключением, быть может, x;
   - n, где n — число;
   - $\mathtt{V}_1\mathtt{V}_2$, где $\mathtt{V}_1$ и $\mathtt{V}_2$ нередуцируемые замкнутые термы, причём $\mathtt{V}_1$ не имеет форму **fun** x → t;
   - $\mathtt{V}_1 \otimes \mathtt{V}_2$, где $\mathtt{V}_1$ и $\mathtt{V}_2$ нередуцируемые замкнутые термы и не являются одновременно числовыми константами;
   - **ifz** $\mathtt{V}_1$ **then** $\mathtt{V}_2$ **else** $\mathtt{V}_3$, где $\mathtt{V}_1$, $\mathtt{V}_2$ и $\mathtt{V}_3$ нередуцируемые замкнутые термы и $\mathtt{V}_1$ не является числом.

(2) Каковы отличия от упражнения 2.6?

Числа и замкнутые термы вида **fun** x → t называются *значениями.*

2.3.3. Вызов по имени

Снова проанализируем редукции, возможные в терме C $\mathtt{b}_1$. Необходимо решить, нужно ли вычислять аргументы функции C до того, как они будут переданы функции, или же их следует передавать функции, не вычисляя.

Стратегия *вызова по имени* требует всякий раз выбирать самый левый редекс, тогда как слабая стратегия вызова по имени требует выбирать самый левый редекс, не входящий в **fun**. Таким образом, C $\mathtt{b}_1$ сводится к 0.

Эта стратегия интересна благодаря следующему свойству, называемому *полнотой*: если терм можно привести к нередуцируемому терму, то редукция с вызовом по имени всегда завершается. Другими словами, $\hookrightarrow_{\mathtt{n}} = \hookrightarrow$. Более того, при вычислении терма `(`**fun** `x` $\to$ `0) (fact 10)` с использованием стратегии вызова по имени, нам не потребуется вычислять факториал `10`. Правда, при вычислении значения терма `(`**fun** `x` $\to$ `x+x) (fact 10)` с той же стратегией, его нужно будет вычислить дважды, поскольку результатом одного шага редукции будет терм `(fact 10) + (fact 10)`. Большинство вычислителей с вызовом по имени стараются в таких случаях избегать дублирования вычислений, запоминая результат первого вычисления. Подобный способ вычислений называется *ленивым*.

2.3.4. Вызов по значению

Вызов по значению, наоборот, предполагает вычисление аргументов до их передачи функции. Он основан на следующем соглашении: терм вида `(`**fun** `x` $\to$ `t) u` редуцируется только в том случае, если `u` является значением. Таким образом, при вычислении `(`**fun** `x` $\to$ `x + x) (fact 10)` нужно начинать с редуцирования аргумента, получая `(`**fun** `x` $\to$ `x + x) 3628800`, а затем уже вычислять самый левый редекс. Это позволит вычислять факториал `10` только один раз.

К этому классу относятся все стратегии, требующие вычисления значения аргументов до их передачи. Такова, к примеру, стратегия вычисления самого левого редекса среди допустимых. Так что вызов по значению является не конкретной стратегией, а целым семейством.

Это соглашение может быть определено и ослаблением правила β-редукции: терм `(`**fun** `x` $\to$ `t) u` считается редексом, только если терм `u` является значением.

Говорят, что слабая стратегия реализует вызов по значению, если она редуцирует термы вида `(`**fun** `x` $\to$ `t) u`, только если `u` является значением и не находится внутри **fun**.

2.3.5. Немного лени не помешает

Даже в случае реализации стратегии вызова по значению условное выражение `ifz` должно вычисляться с вызовом по имени: в терме вида **ifz** `t` **then** `u` **else** `v` никогда не следует вычислять все три аргумента. Вместо этого сначала необходимо вычислить `t`, а затем в зависимости от результата вычислить либо `u`, либо `v`.

Легко видеть, что при вычислении всех трёх аргументов `ifz` вычисление значения `fact 3` не завершается.

Упражнение 2.12. Охарактеризуйте нередуцируемые замкнутые термы

(1) при слабом вызове по имени;

(2) при слабом вызове по значению.

2.4. Операционная семантика с большим шагом

Вместо того, чтобы определять стратегию или ослаблять редукционные правила операционной семантики с малым шагом, можно контролировать порядок вычисления редексов, определив операционную семантику *с большим шагом*.

Операционная семантика с большим шагом для языка программирования даёт индуктивное определение отношения $\hookrightarrow$ без предварительного определения $\longrightarrow$ и $\triangleright$.

2.4.1. Вызов по имени

Начнём с семантики вызова по имени. Рассмотрим терм вида `t u`, который с учётом вызова по имени редуцируется к терму `V`. Будем вычислять те редексы, которые содержатся в `t`, пока не получим нередуцируемый терм. Если это терм вида **fun** `x` $\to$ `t`$'$, то исходный терм редуцируется к терму (**fun** `x` $\to$ `t`$'$`)` `u`, и самым левым редексом оказывается весь терм целиком. Он вычисляется как `(u/x)t`$'$, что в свою очередь редуцируется к `V`. Можно говорить, что терм `t u` сводится при вызове по имени к нередуцируемому терму `V`, если `t` редуцируется к **fun** `x` $\to$ `t`$'$, а `(u/x)t`$'$ к `V`.

Все эти рассуждения можно выразить правилом:

$$\frac{\texttt{t} \hookrightarrow \textbf{fun}\ \texttt{x} \to \texttt{t}' \qquad (\texttt{u/x})\texttt{t}' \hookrightarrow \texttt{V}}{\texttt{t u} \hookrightarrow \texttt{V}},$$

которое будет частью индуктивного определения отношения $\hookrightarrow$ (без предварительного определения $\longrightarrow$ и $\triangleright$).

Другие правила утверждают, что результатом вычисления терма вида **fun** является сам терм, то есть здесь мы определяем отношение слабой редукции:

$$\frac{}{\textbf{fun}\ \texttt{x} \to \texttt{t} \hookrightarrow \textbf{fun}\ \texttt{x} \to \texttt{t}},$$

и что результатом вычисления терма `n` является сам терм:

$$\frac{}{\texttt{n} \hookrightarrow \texttt{n}}.$$

Также есть правило для задания семантики арифметических операций:

$$\frac{\mathtt{u} \hookrightarrow \mathtt{q} \qquad \mathtt{t} \hookrightarrow \mathtt{p}}{\mathtt{t} \otimes \mathtt{u} \hookrightarrow \mathtt{n}} \text{ если } \mathtt{p} \otimes \mathtt{q} = \mathtt{n};$$

два правила для задания семантики условной конструкции `ifz`:

$$\frac{\mathtt{t} \hookrightarrow \mathtt{0} \qquad \mathtt{u} \hookrightarrow \mathtt{V}}{\textbf{ifz } \mathtt{t} \textbf{ then } \mathtt{u} \textbf{ else } \mathtt{v} \hookrightarrow \mathtt{V}},$$

$$\frac{\mathtt{t} \hookrightarrow \mathtt{n} \qquad \mathtt{v} \hookrightarrow \mathtt{V}}{\textbf{ifz } \mathtt{t} \textbf{ then } \mathtt{u} \textbf{ else } \mathtt{v} \hookrightarrow \mathtt{V}} \text{ если } \mathtt{n} \text{ — число, } \mathtt{n} \neq \mathtt{0};$$

правило для оператора неподвижной точки:

$$\frac{(\textbf{fix } \mathtt{x}\ \mathtt{t}/\mathtt{x})\mathtt{t} \hookrightarrow \mathtt{V}}{\textbf{fix } \mathtt{x}\ \mathtt{t} \hookrightarrow \mathtt{V}};$$

и, наконец, правило, определяющее семантику **let**:

$$\frac{(\mathtt{t}/\mathtt{x})\ \mathtt{u} \hookrightarrow \mathtt{V}}{\textbf{let } \mathtt{x} = \mathtt{t} \textbf{ in } \mathtt{u} \hookrightarrow \mathtt{V}}.$$

С помощью структурной индукции по отношению вычисления можно доказать, что результат вычисления терма всегда является значением, то есть либо числом, либо замкнутым термом вида **fun**. Тупиковых термов нет. Вычисление терма `((`**fun** `x` $\to$ `x) 1) 2`, которое в семантике с малым шагом давало тупиковый терм `1 2`, теперь не даёт никакого результата, поскольку ни одно из правил не может быть к нему применено. Действительно, в семантике с большим шагом нет правила, которое объясняло бы как вычислить применение, левая часть которого сводится к числу.

2.4.2. Вызов по значению

Правила, определяющие семантику с вызовом по значению, довольно похожи, за исключением правила для применения: необходимо вычислить значение аргумента до передачи его в функцию:

$$\frac{\mathtt{u} \hookrightarrow \mathtt{W} \qquad \mathtt{t} \hookrightarrow \textbf{fun } \mathtt{x} \to \mathtt{t}' \qquad (\mathtt{W}/\mathtt{x})\mathtt{t}' \hookrightarrow \mathtt{V}}{\mathtt{t}\ \mathtt{u} \hookrightarrow \mathtt{V}},$$

и правила для **let**:

$$\frac{\mathtt{t} \hookrightarrow \mathtt{W} \qquad (\mathtt{W}/\mathtt{x})\mathtt{u} \hookrightarrow \mathtt{V}}{\mathbf{let}\ \mathtt{x} = \mathtt{t}\ \mathbf{in}\ \mathtt{u} \hookrightarrow \mathtt{V}}.$$

Подытоживая, получаем следующие правила:

$$\frac{\mathtt{u} \hookrightarrow \mathtt{W} \qquad \mathtt{t} \hookrightarrow \mathbf{fun}\ \mathtt{x} \to \mathtt{t}' \qquad (\mathtt{W}/\mathtt{x})\mathtt{t}' \hookrightarrow \mathtt{V}}{\mathtt{t}\ \mathtt{u} \hookrightarrow \mathtt{V}},$$

$$\frac{}{\mathbf{fun}\ \mathtt{x} \to \mathtt{t} \hookrightarrow \mathbf{fun}\ \mathtt{x} \to \mathtt{t}},$$

$$\frac{}{\mathtt{n} \hookrightarrow \mathtt{n}},$$

$$\frac{\mathtt{u} \hookrightarrow \mathtt{q} \qquad \mathtt{t} \hookrightarrow \mathtt{p}}{\mathtt{t} \otimes \mathtt{u} \hookrightarrow \mathtt{n}} \text{ если } \mathtt{p} \otimes \mathtt{q} = \mathtt{n},$$

$$\frac{\mathtt{t} \hookrightarrow \mathtt{0} \qquad \mathtt{u} \hookrightarrow \mathtt{V}}{\mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v} \hookrightarrow \mathtt{V}},$$

$$\frac{\mathtt{t} \hookrightarrow \mathtt{n} \qquad \mathtt{v} \hookrightarrow \mathtt{V}}{\mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v} \hookrightarrow \mathtt{V}} \text{ если } \mathtt{n} \text{ — число, } \mathtt{n} \neq \mathtt{0},$$

$$\frac{(\mathbf{fix}\ \mathtt{x}\ \mathtt{t}/\mathtt{x})\mathtt{t} \hookrightarrow \mathtt{V}}{\mathbf{fix}\ \mathtt{x}\ \mathtt{t} \hookrightarrow \mathtt{V}},$$

$$\frac{\mathtt{t} \hookrightarrow \mathtt{W} \qquad (\mathtt{W}/\mathtt{x})\mathtt{u} \hookrightarrow \mathtt{V}}{\mathbf{let}\ \mathtt{x} = \mathtt{t}\ \mathbf{in}\ \mathtt{u} \hookrightarrow \mathtt{V}}.$$

Заметим, что даже при вызове по значению правила для `ifz` сохраняются:

$$\frac{\mathtt{t} \hookrightarrow \mathtt{0} \qquad \mathtt{u} \hookrightarrow \mathtt{V}}{\mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v} \hookrightarrow \mathtt{V}},$$

$$\frac{\mathtt{t} \hookrightarrow \mathtt{n} \qquad \mathtt{v} \hookrightarrow \mathtt{V}}{\mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v} \hookrightarrow \mathtt{V}} \text{ если } \mathtt{n} \text{ — число, } \mathtt{n} \neq \mathtt{0},$$

то есть второй и третий аргументы `ifz` не вычисляются до тех пор, пока они не потребуются.

Заметим также, что при вызове по значению сохраняется и правило

для **fix**:

$$\frac{\texttt{(fix x t/x)t} \hookrightarrow \texttt{V}}{\textbf{fix}\ \texttt{x t} \hookrightarrow \texttt{V}}.$$

Необходимо устоять перед искушением вычислить терм **fix** `x t` до значения `W` перед его подстановкой в `t`, так как правило

$$\frac{\textbf{fix}\ \texttt{x t} \hookrightarrow \texttt{W} \qquad \texttt{(W/x)t} \hookrightarrow \texttt{V}}{\textbf{fix}\ \texttt{x t} \hookrightarrow \texttt{V}}$$

требует для вычисления **fix** `x t` начать с вычисления **fix** `x t`, что приведёт к появлению цикла, и терм `fact 3` никогда не вычислится до значения — его редуцирование запустит бесконечный вычислительный процесс.

Заметим, наконец, что возможны и другие комбинации правил. Например, некоторые варианты семантики с вызовом по имени используют в правиле **let** вызов по значению.

Упражнение 2.13. Каковы значения следующих PCF-термов согласно операционной семантике с большим шагом?

(1) `(`**fun** `x` $\to$ **fun** `x` $\to$ `x) 2 3`;

(2) `(`**fun** `x` $\to$ **fun** `y` $\to$ `((`**fun** `x` $\to$ `(x + y)) x)) 5 4`.

Сравните ответы с упражнением 2.7.

Упражнение 2.14. Каково значение терма

let `x = 4` **in let** `f =` **fun** `y` $\to$ `y + x` **in let** `x = 5` **in** `f 6`

с точки зрения операционной семантики с большим шагом, 10 или 11? Сравните ответ с упражнением 2.8.

2.5. Вычисление PCF-программ

Вычислителем PCF называется программа, которая принимает замкнутый PCF-терм и возвращает его значение. При чтении снизу вверх правила семантики с большим шагом можно рассматривать как ядро такого вычислителя: для вычисления применения `t u` нужно начать с вычисления `u` и `t` и т. д. Это легко выразить в языке типа Caml:

```
let rec eval p = match p with
  | App(t,u) -> let w = eval u
                in let v = eval t
                in ...
  | ...
```

В случае с применением правила операционной семантики с большим шагом оставляют нам некоторую свободу в порядке вычисления термов `t` или `u` — вызов по значению является не стратегией, а семейством стратегий — однако терм $\texttt{(W/x)t}'$ в любом случае должен вычисляться третьим, поскольку он строится на основе результатов вычисления первых двух.

Упражнение 2.15.

(1) Напишите вычислитель PCF, то есть программу, которая принимает на вход замкнутый терм и вычисляет его значение, с вызовом по имени.

(2) Напишите вычислитель с вызовом по значению.

(3) Вычислите значения термов `fact 6` и $\texttt{C b}_1$ с помощью обоих вычислителей.

Денотационную семантику для PCF определить сложнее. Это может показаться парадоксальным, ведь PCF является функциональным языком, а значит, его программы должны легко интерпретироваться как функции. Проблема, однако, в том, что PCF допускает применение любого объекта к любому, и ничто не мешает нам, к примеру, записать терм $\textbf{fun}\ \texttt{x} \to \texttt{(x x)}$. В отличие от математических функций функции PCF не имеют области определения. По этим причинам денотационная семантика будет дана в главе 5, то есть уже после того, как в язык будут добавлены типы.

Глава 3. От вычисления к интерпретации

3.1. Вызов по имени

Применяя операционную семантику с большим шагом, можно построить вычислитель для языка PCF, в котором терм вида `(`**fun** `x` $\to$ `t) u` начинает вычисляться с помощью подстановки терма `u` в тело функции `t` всюду вместо переменной `x`. Например, чтобы вычислить терм `(`**fun** `x` $\to$ `(x*x) + x) 4`, подставим `4` вместо `x` в терме `(x*x) + x`, а затем вычислим полученный терм `(4*4) + 4`. Подстановка является затратной операцией; чтобы повысить производительность вычислителя, можно было бы вместо неё хранить ассоциацию `x = 4` в отдельной структуре, называемой *окружением*, и вычислять терм `(x*x) + x` в этом окружении. Программа, вычисляющая термы таким способом, называется *интерпретатором*.

Окружение это функция с конечной областью определения, действующая из переменных в термы. По сути, это то же самое, что подстановка, но в иных обозначениях. Окружение записывается в виде списка пар $\mathtt{x}_1 = \mathtt{t}_1, \ldots, \mathtt{x}_n = \mathtt{t}_n$, причём одна и та же переменная `x` может встречаться несколько раз, и в этом случае самая правая пара имеет приоритет. Таким образом, в окружении `x = 3, y = 4, x = 5, z = 8` используется значение `x = 5`, но не `x = 3`, которое, как говорят, *скрыто* парой `x = 5`. Наконец, если `e` это окружение и `x = t` — пара, через `e, x = t` обозначается список, полученный расширением `e` с помощью `x = t`.

Во время вычисления терма можно встретить свободную переменную `x`. В этом случае происходит поиск ассоциированного с ней терма в окружении. Можно показать, что если начать с замкнутого терма, такой поиск всегда будет завершаться успехом.

Фактически ситуация несколько более сложна, так как в дополнение к терму `u`, ассоциированному с переменной в окружении, необходимо также найти окружение, ассоциированное с `u`. Пара: терм и окружение — называется *задумкой* (thunk) и будет записываться $\langle \mathtt{u}, \mathtt{e} \rangle$.

Аналогично при интерпретации терма вида **fun** `x` $\to$ `t` в окружении `e` результат не может быть просто термом **fun** `x` $\to$ `t`, потому что может содержать свободные переменные, и во время интерпретации терма `t` понадобятся задумки, ассоциированные с этими переменными в окружении `e`. Расширим понятие значения, добавив к числам *замыкания*, состоящие из

терма, который *должен иметь вид* **fun** x → t, и окружения e. Будем записывать такие значения следующим образом: ⟨x, t, e⟩. Значения больше не являются подмножеством термов, и возникает необходимость определить язык значений независимо от языка термов.

Как следствие, придётся переписать правила операционной семантики с большим шагом при вызове по имени для PCF, чтобы ввести отношение вида e ⊢ t ↪ V, читается «t интерпретируется как V в окружении e», где e обозначает окружение, t — терм, V — значение. Когда окружение e пусто, это отношение записывается так: ⊢ t ↪ V. К правилам, расширяющим окружение, относятся применение, которое добавляет пару, состоящую из переменной x и задумки ⟨u, e⟩, правило для **let**, которое добавляет пару, состоящую из переменной x и задумки ⟨t, e⟩, и правило для **fix**, которое добавляет пару, состоящую из переменной x и задумки ⟨**fix** x t, e⟩. В последнем правиле терм t дублируется: одна из копий интерпретируется, а другая остаётся в окружении для использования из рекурсивных вызовов, возникающих при вычислении первой.

$$\frac{e' \vdash t \hookrightarrow V}{e \vdash x \hookrightarrow V} \text{ если e содержит } x = \langle t, e' \rangle,$$

$$\frac{e \vdash t \hookrightarrow \langle x, t', e' \rangle \qquad (e', x = \langle u, e \rangle) \vdash t' \hookrightarrow V}{e \vdash t\ u \hookrightarrow V},$$

$$\frac{}{e \vdash \textbf{fun}\ x \to t \hookrightarrow \langle x, t, e \rangle},$$

$$\frac{}{e \vdash n \hookrightarrow n},$$

$$\frac{e \vdash u \hookrightarrow q \qquad e \vdash t \hookrightarrow p}{e \vdash t \otimes u \hookrightarrow n} \text{ если } p \otimes q = n,$$

$$\frac{e \vdash t \hookrightarrow 0 \qquad e \vdash u \hookrightarrow V}{e \vdash \textbf{ifz}\ t\ \textbf{then}\ u\ \textbf{else}\ v \hookrightarrow V},$$

$$\frac{e \vdash t \hookrightarrow n \qquad e \vdash v \hookrightarrow V}{e \vdash \textbf{ifz}\ t\ \textbf{then}\ u\ \textbf{else}\ v \hookrightarrow V} \text{ если n — число, } n \neq 0,$$

$$\frac{(e, x = \langle \textbf{fix}\ x\ t, e \rangle) \vdash t \hookrightarrow V}{e \vdash \textbf{fix}\ x\ t \hookrightarrow V},$$

$$\frac{(e, x = \langle t, e \rangle) \vdash u \hookrightarrow V}{e \vdash \textbf{let}\ x = t\ \textbf{in}\ u \hookrightarrow V}.$$

Упражнение 3.1. Запрограммируйте интерпретатор PCF с вызовом по имени.

Упражнение 3.2. Каковы значения следующих термов согласно приведённым выше правилам интерпретации для PCF?

(1) `(`**fun** `x` $\to$ **fun** `x` $\to$ `x) 2 3`;

(2) `(`**fun** `x` $\to$ **fun** `y` $\to$ `((`**fun** `x` $\to$ `(x + y)) x)) 5 4`.

Сравните ответы с упражнениями 2.7 и 2.13.

Упражнение 3.3. Каково значение терма

let `x = 4` **in let** `f =` **fun** `y` $\to$ `y + x` **in let** `x = 5` **in** `f 6`

с точки зрения правил интерпретации для PCF, 10 или 11? Сравните ответ с упражнениями 2.8 и 2.14.

3.2. Вызов по значению

Вариант с семантикой вызова по значению проще. Действительно, при интерпретации терма `(`**fun** `x` $\to$ `t) u` сначала интерпретируется терм `u`. Результатом является значение, то есть число или замыкание, достаточно лишь связать переменную `x` в окружении с этим значением. Аналогично при интерпретации терма **let** `x = t` **in** `u` вначале интерпретируется терм `t`. Результатом является значение, и вновь достаточно связать переменную `x` в окружении с этим значением. Таким образом, окружения будут сопоставлять переменным значения, а не задумки (которые заморожены до момента интерпретации). Исчезает необходимость в понятии задумки.

Однако вычислительное правило для **fix**, в отличие от правил для применения или **let**, требует подстановки вместо переменной терма в форме **fix** `x t`, который не является значением, и попытка получения значения которого перед подстановкой или сохранением в окружении привела бы к бесконечным вычислениям (как упоминалось выше). В окружение придётся включить *оснащённые значения*, которые являются либо значениями, либо задумками, содержащими терм вида **fix** `x t` и окружение `e`. При обращении к такому оснащённому значению понадобится интерпретировать его, если оно представляет из себя задумку. Это приводит к следующему набору правил:

$$\frac{}{\texttt{e} \vdash \texttt{x} \hookrightarrow \texttt{V}}$$ если `e` содержит `x = V`,

$$\frac{\mathtt{e}' \vdash \mathbf{fix}\ \mathtt{y}\ \mathtt{t} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \mathtt{x} \hookrightarrow \mathtt{V}}\ \text{если } \mathtt{e} \text{ содержит } \mathtt{x} = \langle \mathbf{fix}\ \mathtt{y}\ \mathtt{t}, \mathtt{e}' \rangle,$$

$$\frac{\mathtt{e} \vdash \mathtt{u} \hookrightarrow \mathtt{W} \qquad \mathtt{e} \vdash \mathtt{t} \hookrightarrow \langle \mathtt{x}, \mathtt{t}', \mathtt{e}' \rangle \qquad (\mathtt{e}', \mathtt{x} = \mathtt{W}) \vdash \mathtt{t}' \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \mathtt{t}\ \mathtt{u} \hookrightarrow \mathtt{V}},$$

$$\frac{}{\mathtt{e} \vdash \mathbf{fun}\ \mathtt{x} \to \mathtt{t} \hookrightarrow \langle \mathtt{x}, \mathtt{t}, \mathtt{e} \rangle},$$

$$\frac{}{\mathtt{e} \vdash \mathtt{n} \hookrightarrow \mathtt{n}},$$

$$\frac{\mathtt{e} \vdash \mathtt{u} \hookrightarrow \mathtt{q} \qquad \mathtt{e} \vdash \mathtt{t} \hookrightarrow \mathtt{p}}{\mathtt{e} \vdash \mathtt{t} \otimes \mathtt{u} \hookrightarrow \mathtt{n}}\ \text{если } \mathtt{p} \otimes \mathtt{q} = \mathtt{n},$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \hookrightarrow \mathtt{0} \qquad \mathtt{e} \vdash \mathtt{u} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v} \hookrightarrow \mathtt{V}},$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \hookrightarrow \mathtt{n} \qquad \mathtt{e} \vdash \mathtt{v} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v} \hookrightarrow \mathtt{V}}\ \text{если } \mathtt{n} \text{ — число, } \mathtt{n} \neq \mathtt{0},$$

$$\frac{(\mathtt{e}, \mathtt{x} = \langle \mathbf{fix}\ \mathtt{x}\ \mathtt{t}, \mathtt{e} \rangle) \vdash \mathtt{t} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \mathbf{fix}\ \mathtt{x}\ \mathtt{t} \hookrightarrow \mathtt{V}},$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \hookrightarrow \mathtt{W} \qquad (\mathtt{e}, \mathtt{x} = \mathtt{W}) \vdash \mathtt{u} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \mathbf{let}\ \mathtt{x} = \mathtt{t}\ \mathbf{in}\ \mathtt{u} \hookrightarrow \mathtt{V}}.$$

Упражнение 3.4. При вычислении `(fact 3)`, где функция `fact` определена следующим образом: **fix** `f` **fun** `n` → **ifz** `n` **then** `1` **else** `n * (f (n - 1))`, начинать следует с рекурсивного вызова функции `fact` для аргумента `2`. После возвращения из рекурсивного вызова для вычисления значения `n` и умножения переменная `n` ассоциирована со значением `2` или `3`? Почему?

Упражнение 3.5. Запрограммируйте интерпретатор PCF с вызовом по значению.

3.3. Оптимизация: индексы де Брауна

В правилах для операционной семантики с большим шагом окружение представляет из себя список пар, состоящих из переменной и оснащённого значения. Можно заменить эту структуру на пару списков той же длины, один из которых содержит переменные, а другой — значения. Так, например, список `x = 12`, `y = 14`, `z = 16`, `w = 18`, может быть заменён на список

переменных `x, y, z, w` и список оснащённых значений `12`, `14`, `16`, `18`. Для поиска оснащённого значения, соответствующего переменной, необходимо просмотреть первый список для определения позиции переменной, а затем найти в другом списке элемент на этой позиции. Позиция переменной в первом списке это число, называемое *индексом де Брауна* переменной в окружении. В общем случае можно связать число `0` с последним («самым правым») элементом списка, число `1` — с предшествующим, ..., число `n - 1` — с первым («самым левым») элементом списка.

Список переменных, которые понадобятся для интерпретации каждого подтерма, может быть вычислен до начала процесса интерпретации. Фактически, перед тем как интерпретировать терм, можно поставить в соответствие каждому вхождению переменной её индекс де Брауна. Например, если интерпретируется терм **fun** `x` → **fun** `y` → `(x + (fun z` → **fun** `w` → `(x + y + z + w))(2 * 8)(14 + 4))(5 + 7)(20 - 6)`, переменная `y` неизбежно будет интерпретирована в окружении вида `x = ., y = ., z = ., w = .`, то есть, чтобы найти значение, соответствующее `y`, нужно взять значение по индексу `2`. Можно приписать этот индекс переменной с самого начала.

Чтобы вычислить индексы де Брауна, нужно просто обойти весь терм, поддерживая *окружение переменных*, то есть список переменных, в котором индекс `p` приписывается переменной `x` в окружении `e`, если `p` обозначает позицию переменной `x` в окружении `e`, считая с конца.

— $|\mathtt{x}|_{\mathtt{e}} = \mathtt{x}^{\mathtt{p}}$, где `p` это позиция `x` в окружении `e`,

— $|\mathtt{t\ u}|_{\mathtt{e}} = |\mathtt{t}|_{\mathtt{e}}\ |\mathtt{u}|_{\mathtt{e}}$,

— $|\textbf{fun}\ \mathtt{x} \to \mathtt{t}|_{\mathtt{e}} = \textbf{fun}\ \mathtt{x} \to |\mathtt{t}|_{\mathtt{e},\mathtt{x}}$,

— $|\mathtt{n}|_{\mathtt{e}} = \mathtt{n}$,

— $|\mathtt{t} + \mathtt{u}|_{\mathtt{e}} = |\mathtt{t}|_{\mathtt{e}} + |\mathtt{u}|_{\mathtt{e}}$,

— $|\mathtt{t} - \mathtt{u}|_{\mathtt{e}} = |\mathtt{t}|_{\mathtt{e}} - |\mathtt{u}|_{\mathtt{e}}$,

— $|\mathtt{t} * \mathtt{u}|_{\mathtt{e}} = |\mathtt{t}|_{\mathtt{e}} * |\mathtt{u}|_{\mathtt{e}}$,

— $|\mathtt{t}/\mathtt{u}|_{\mathtt{e}} = |\mathtt{t}|_{\mathtt{e}}/|\mathtt{u}|_{\mathtt{e}}$,

— $|\textbf{ifz}\ \mathtt{t}\ \textbf{then}\ \mathtt{u}\ \textbf{else}\ \mathtt{v}|_{\mathtt{e}} = \textbf{ifz}\ |\mathtt{t}|_{\mathtt{e}}\ \textbf{then}\ |\mathtt{u}|_{\mathtt{e}}\ \textbf{else}\ |\mathtt{v}|_{\mathtt{e}}$,

— $|\textbf{fix}\ \mathtt{x}\ \mathtt{t}|_{\mathtt{e}} = \textbf{fix}\ \mathtt{x}\ |\mathtt{t}|_{\mathtt{e},\mathtt{x}}$,

— $|\textbf{let}\ \mathtt{x} = \mathtt{t}\ \textbf{in}\ \mathtt{u}|_{\mathtt{e}} = \textbf{let}\ \mathtt{x} = |\mathtt{t}|_{\mathtt{e}}\ \textbf{in}\ |\mathtt{u}|_{\mathtt{e},\mathtt{x}}$.

Например, терм выше будет записан так: **fun** `x` → **fun** `y` → $(\mathtt{x}^1 + (\textbf{fun}\ \mathtt{z} \to \textbf{fun}\ \mathtt{w} \to (\mathtt{x}^3 + \mathtt{y}^2 + \mathtt{z}^1 + \mathtt{w}^0))(2 * 8)(14 + 4))(5 + 7)(20 - 6)$.

Нетрудно показать, что вхождение подтерма, преобразованное в окружение переменных $\mathtt{x}_1, \ldots, \mathtt{x}_n$, всегда будет интерпретировано в окружении

вида $\mathtt{x_1 = .}, \ldots, \mathtt{x_n = .}$, поэтому для отыскания значения переменной с индексом p достаточно взять значение p-го элемента окружения.

Это подсказывает другой путь интерпретации терма: вначале вычисляются индексы де Брауна для каждого вхождения переменной; как только все индексы известны, необходимость в хранении списка переменных в окружении отпадает. Окружение будет содержать лишь список оснащённых значений. Аналогично удаляются имена переменных из замыканий и задумок. Действительно, теперь имена переменных бесполезны, пример выше можно было бы записать так: **fun** _ → **fun** _ → (_^{1} + (**fun** _ → **fun** _ → (_^{3} + _^{2} + _^{1} + _^{0}))(2 * 8)(14 + 4))(5 + 7)(20 - 6).

Правила операционной семантики с большим шагом могут быть теперь определены следующим образом:

$$\frac{}{\mathtt{e} \vdash \_^{\mathtt{p}} \hookrightarrow \mathtt{V}}$$ если V это p-ый элемент e,

$$\frac{\mathtt{e}' \vdash \mathbf{fix}\ \_\ \mathtt{t} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \_^{\mathtt{p}} \hookrightarrow \mathtt{V}}$$ если p-ый элемент e это $\langle \mathbf{fix}\ \_\ \mathtt{t}, \mathtt{e}' \rangle$,

$$\frac{\mathtt{e} \vdash \mathtt{u} \hookrightarrow \mathtt{W} \qquad \mathtt{e} \vdash \mathtt{t} \hookrightarrow \langle \mathtt{t}', \mathtt{e}' \rangle \qquad (\mathtt{e}', \mathtt{W}) \vdash \mathtt{t}' \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \mathtt{t}\ \mathtt{u} \hookrightarrow \mathtt{V}},$$

$$\frac{}{\mathtt{e} \vdash \mathbf{fun}\ \_ \to \mathtt{t} \hookrightarrow \langle \mathtt{t}, \mathtt{e} \rangle},$$

$$\frac{}{\mathtt{e} \vdash \mathtt{n} \hookrightarrow \mathtt{n}},$$

$$\frac{\mathtt{e} \vdash \mathtt{u} \hookrightarrow \mathtt{q} \qquad \mathtt{e} \vdash \mathtt{t} \hookrightarrow \mathtt{p}}{\mathtt{e} \vdash \mathtt{t} \otimes \mathtt{u} \hookrightarrow \mathtt{n}}$$ если $\mathtt{p} \otimes \mathtt{q} = \mathtt{n}$,

$$\frac{\mathtt{e} \vdash \mathtt{t} \hookrightarrow 0 \qquad \mathtt{e} \vdash \mathtt{u} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v} \hookrightarrow \mathtt{V}},$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \hookrightarrow \mathtt{n} \qquad \mathtt{e} \vdash \mathtt{v} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v} \hookrightarrow \mathtt{V}}$$ если n — число, $\mathtt{n} \neq 0$,

$$\frac{(\mathtt{e}, \langle \mathbf{fix}\ \_\ \mathtt{t}, \mathtt{e} \rangle) \vdash \mathtt{t} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \mathbf{fix}\ \_\ \mathtt{t} \hookrightarrow \mathtt{V}},$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \hookrightarrow \mathtt{W} \qquad (\mathtt{e}, \mathtt{W}) \vdash \mathtt{u} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \mathbf{let}\ \_ = \mathtt{t}\ \mathbf{in}\ \mathtt{u} \hookrightarrow \mathtt{V}}.$$

Упражнение 3.6. Напишите программу, заменяющую каждую переменную на её индекс де Брауна, и интерпретатор полученного языка.

Упражнение 3.7. Запишите правила операционной семантики с большим шагом для вызова по имени, используя индексы де Брауна.

Преимущества такой записи, избавляющей от имён переменных, будут подчёркнуты в следующей главе при изучении компиляции.

Заметим, между прочим, что два терма имеют один и тот же вид при записи с помощью индексов де Брауна в том и только том случае, если они α-эквивалентны. Этот факт даёт ещё одно определение алфавитной эквивалентности. Замена переменных на индексы, обозначающие место их связывания, снова приводит к важной мысли о том, что связанные переменные это всего лишь «заглушки».

3.4. Построение функций с помощью неподвижных точек

В большинстве языков программирования рекурсия допускается лишь при определении функций. Конструкция **fix** применима только для терма вида **fun**, поэтому можно было бы заменить символ **fix** на **fixfun** `f x` $\to$ `t`, который связывает в своём аргументе сразу две переменные. Семантическое правило с большим шагом при вызове по значению для последнего может быть выведено из приведённых выше правил для **fix** и **fun**:

$$\frac{}{\texttt{e} \vdash \textbf{fixfun}\ \texttt{f x} \to \texttt{t} \hookrightarrow \langle \texttt{x, t, (e, f =} \langle \textbf{fixfun}\ \texttt{f x} \to \texttt{t, e} \rangle \texttt{)} \rangle}\ .$$

В этом случае можно определить более простые версии правил для интерпретатора с вызовом по значению.

3.4.1. Первая версия: рекурсивные замыкания

Будем выделять замыкания вида $\langle \texttt{x, t, (e, f =} \langle \textbf{fixfun}\ \texttt{f x} \to \texttt{t, e} \rangle \texttt{)} \rangle$, которые станем записывать в виде $\langle \texttt{f, x, t, e} \rangle$ и называть *рекурсивными замыканиями*.

Правило, данное для интерпретации конструкции **fixfun** `f x` $\to$ `t`, может быть переформулировано следующим образом:

$$\frac{}{\texttt{e} \vdash \textbf{fixfun}\ \texttt{f x} \to \texttt{t} \hookrightarrow \langle \texttt{f, x, t, e} \rangle}\ .$$

При попытке интерпретации применения `t u` в семантике вызова по значению, если терм `t` интерпретирован как рекурсивное замыкание $\langle \texttt{f, x, t}', \texttt{e}' \rangle$, то есть $\langle \texttt{x, t}', \texttt{(e}', \texttt{f =} \langle \textbf{fixfun}\ \texttt{f x} \to \texttt{t}', \texttt{e}' \rangle \texttt{)} \rangle$, а терм `u` —

как значение W, требуется интерпретировать терм $\mathtt{t}'$ в окружении $\mathtt{e}', \mathtt{f} = \langle \textbf{fixfun}\ \mathtt{f}\ \mathtt{x} \to \mathtt{t}', \mathtt{e}' \rangle, \mathtt{x} = \mathtt{W}$.

Можно ожидать интерпретации задумки $\langle \textbf{fixfun}\ \mathtt{f}\ \mathtt{x} \to \mathtt{t}', \mathtt{e}' \rangle$, которая возникает в таком окружении, что приведёт к использованию правила **fixfun**, являющегося рекурсивным замыканием $\langle \mathtt{f}, \mathtt{x}, \mathtt{t}', \mathtt{e}' \rangle$. В случае рекурсивных замыканий правило для применения может быть специализировано следующим образом:

$$\frac{\begin{array}{c}\mathtt{e} \vdash \mathtt{u} \hookrightarrow \mathtt{W} \\ \mathtt{e} \vdash \mathtt{t} \hookrightarrow \langle \mathtt{f}, \mathtt{x}, \mathtt{t}', \mathtt{e}' \rangle \\ (\mathtt{e}', \mathtt{f} = \langle \mathtt{f}, \mathtt{x}, \mathtt{t}', \mathtt{e}' \rangle, \mathtt{x} = \mathtt{W}) \vdash \mathtt{t}' \hookrightarrow \mathtt{V}\end{array}}{\mathtt{e} \vdash \mathtt{t}\ \mathtt{u} \hookrightarrow \mathtt{V}}.$$

В этом правиле не используются задумки; таким образом, при вызове по значению благодаря введению рекурсивных замыканий исчезают задумки, также отпадает необходимость в правиле их интерпретации.

Последнее упрощение: обычные замыкания $\langle \mathtt{x}, \mathtt{t}, \mathtt{e} \rangle$ могут быть заменены рекурсивными $\langle \mathtt{f}, \mathtt{x}, \mathtt{t}, \mathtt{e} \rangle$, где f это произвольная переменная, которая не входит в t. Тогда можно удалить правило применения для случая обычных замыканий.

Окончательно мы получаем следующие правила:

$$\frac{}{\mathtt{e} \vdash \mathtt{x} \hookrightarrow \mathtt{V}}\ \text{если e содержит } \mathtt{x} = \mathtt{V},$$

$$\frac{\begin{array}{c}\mathtt{e} \vdash \mathtt{u} \hookrightarrow \mathtt{W} \\ \mathtt{e} \vdash \mathtt{t} \hookrightarrow \langle \mathtt{f}, \mathtt{x}, \mathtt{t}', \mathtt{e}' \rangle \\ (\mathtt{e}', \mathtt{f} = \langle \mathtt{f}, \mathtt{x}, \mathtt{t}', \mathtt{e}' \rangle, \mathtt{x} = \mathtt{W}) \vdash \mathtt{t}' \hookrightarrow \mathtt{V}\end{array}}{\mathtt{e} \vdash \mathtt{t}\ \mathtt{u} \hookrightarrow \mathtt{V}},$$

$$\frac{}{\mathtt{e} \vdash \textbf{fun}\ \mathtt{x} \to \mathtt{t} \hookrightarrow \langle \mathtt{f}, \mathtt{x}, \mathtt{t}, \mathtt{e} \rangle},$$

где f это произвольная переменная, отличная от x и не входящая ни в t, ни в e,

$$\frac{}{\mathtt{e} \vdash \textbf{fixfun}\ \mathtt{f}\ \mathtt{x} \to \mathtt{t} \hookrightarrow \langle \mathtt{f}, \mathtt{x}, \mathtt{t}, \mathtt{e} \rangle},$$

$$\frac{}{\mathtt{e} \vdash \mathtt{n} \hookrightarrow \mathtt{n}},$$

$$\frac{\mathtt{e} \vdash \mathtt{u} \hookrightarrow \mathtt{q} \qquad \mathtt{e} \vdash \mathtt{t} \hookrightarrow \mathtt{p}}{\mathtt{e} \vdash \mathtt{t} \otimes \mathtt{u} \hookrightarrow \mathtt{n}}\ \text{если } \mathtt{p} \otimes \mathtt{q} = \mathtt{n},$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \hookrightarrow \mathtt{0} \qquad \mathtt{e} \vdash \mathtt{u} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \textbf{ifz } \mathtt{t} \textbf{ then } \mathtt{u} \textbf{ else } \mathtt{v} \hookrightarrow \mathtt{V}},$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \hookrightarrow \mathtt{n} \qquad \mathtt{e} \vdash \mathtt{v} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \textbf{ifz } \mathtt{t} \textbf{ then } \mathtt{u} \textbf{ else } \mathtt{v} \hookrightarrow \mathtt{V}} \text{ если } \mathtt{n} \text{ — число, } \mathtt{n} \neq \mathtt{0},$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \hookrightarrow \mathtt{W} \qquad (\mathtt{e}, \mathtt{x} = \mathtt{W}) \vdash \mathtt{u} \hookrightarrow \mathtt{V}}{\mathtt{e} \vdash \textbf{let } \mathtt{x} = \mathtt{t} \textbf{ in } \mathtt{u} \hookrightarrow \mathtt{V}}.$$

Упражнение 3.8. Запрограммируйте интерпретатор PCF с вызовом по значению, использующий рекурсивные замыкания.

Упражнение 3.9. Как изменятся правила операционной семантики с большим шагом, использующие рекурсивные замыкания, если заменить переменные на индексы де Брауна (см. раздел 3.3)?

3.4.2. Вторая версия: рациональные значения

В правиле

$$\frac{}{\mathtt{e} \vdash \textbf{fixfun } \mathtt{f}\ \mathtt{x} \to \mathtt{t} \hookrightarrow \langle \mathtt{x}, \mathtt{t}, (\mathtt{e}, \mathtt{f} = \langle \textbf{fixfun } \mathtt{f}\ \mathtt{x} \to \mathtt{t}, \mathtt{e} \rangle) \rangle}$$

можно ожидать будущую интерпретацию задумки $\langle \textbf{fixfun } \mathtt{f}\ \mathtt{x} \to \mathtt{t}, \mathtt{e} \rangle$. Конечно, её значением является терм $\langle \mathtt{x}, \mathtt{t}, (\mathtt{e}, \mathtt{f} = \langle \textbf{fixfun } \mathtt{f}\ \mathtt{x} \to \mathtt{t}, \mathtt{e} \rangle) \rangle$, в котором эта задумка возникает вновь. Можно было бы пытаться интерпретировать её снова и снова.

Как было сказано ранее, этот вид интерпретации терма в форме **fix** `f t` до подстановки или сохранения в окружение приводит к бесконечным вычислениям. В данном случае это приводит к построению бесконечного значения $\langle \mathtt{x}, \mathtt{t}, (\mathtt{e}, \mathtt{f} = \langle \mathtt{x}, \mathtt{t}, (\mathtt{e}, \mathtt{f} = \langle \mathtt{x}, \mathtt{t}, (\mathtt{e}, \mathtt{f} = \langle \mathtt{x}, \mathtt{t}, (\mathtt{e}, \mathtt{f} = \ldots) \rangle) \rangle) \rangle) \rangle) \rangle$, которое представляет из себя бесконечный, но рациональный терм. Существуют хорошо известные способы представления рациональных деревьев в памяти компьютера. В нашем случае такое значение можно представить структурой, изображённой на рис. 1.

Используя обозначение `FIX X` $\langle \mathtt{x}, \mathtt{t}, (\mathtt{e}, \mathtt{f} = \mathtt{X}) \rangle$ для этого рационального значения, правило выше можно записать так:

$$\frac{}{\mathtt{e} \vdash \textbf{fixfun } \mathtt{f}\ \mathtt{x} \to \mathtt{t} \hookrightarrow \mathtt{FIX\ X}\ \langle \mathtt{x}, \mathtt{t}, (\mathtt{e}, \mathtt{f} = \mathtt{X}) \rangle}.$$

Здесь снова не возникает необходимости в задумках.

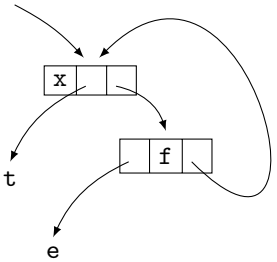


Рис. 1: представление рационального терма

Заметим, что иногда лучше представлять такое рациональное значение эквивалентным способом, изображённым на рис. 2. В этом случае следовало бы говорить о рациональных окружениях.

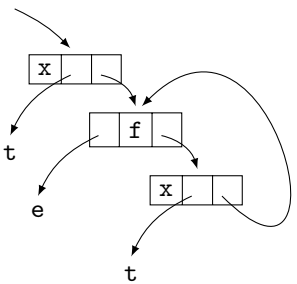


Рис. 2: эквивалентное представление рационального терма

Упражнение 3.10. Запрограммируйте интерпретатор PCF с вызовом по значению, используя рациональные значения.

Упражнение 3.11. Как выполнить такое преобразование семантических правил для операционной семантики с большим шагом, если заменить переменные их индексами де Брауна (см. раздел 3.3)?

Упражнение 3.12. Возможно ли использовать технику рациональных значений для разработки интерпретатора полного PCF, то есть такого, в котором с помощью неподвижных точек определяются не только функции, но и произвольные объекты? Подсказка: каково рациональное представление значения терма **fix** `x x`?

Подведём итоги. В этом разделе было показано, что если переменная `x` входит в терм `t`, то правило редукции **fix** `x t` $\longrightarrow$ `(`**fix** `x t/x)t` может применяться к терму **fix** `x t` бесконечно, потому что терм `(`**fix** `x t/x)t` вновь содержит терм **fix** `x t` в качестве подтерма. В рекурсивном определении `f = G(f)` это соответствовало бы подстановке `G(f)` вместо `f` бесконечное число раз, что приводит к бесконечной программе `f = G(G(G(...)))`. В каком-то смысле это объясняет интуитивное представление о том, что рекурсивные программы это бесконечные программы. Например, терм `fact` мог бы быть записан так:

ifz `x` **then** `1` **else** `x*(`**ifz** `x - 1` **then** `1` **else** `(x - 1) *`
`(`**ifz** `x - 2` **then** `1` **else** `(x - 2) *...)).`

Такая замена должна производиться только по необходимости — в ленивом стиле.

Было рассмотрено несколько способов отразить это поведение в семантике PCF — и, в конце концов, в коде интерпретатора PCF: подставить **fix** `x t` вместо `x` и «заморозить» этот терм, если он оказался внутри **fun** или **ifz**, сохранить этот редекс в виде задумки или рекурсивного замыкания и выполнять его только по необходимости, представить терм `f = G(G(G(...)))` как рациональное дерево и обходить его только по необходимости. Наконец, ещё один способ основывается на кодировании **fix**, приведённом в упражнении 2.10, и состоит в том, чтобы редуцировать соответствующий терм (который предполагает дублирование подтерма) только по необходимости.

Упражнение 3.13 (расширение: PCF с парами)**.** Дополним PCF конструкциями следующего вида: `t, u`, представляющими пары, в которых первая компонента это `t`, а вторая — `u`; если `t` это пара, то `fst t` и `snd t` это первая и вторая компоненты `t` соответственно. Запишите семантические правила с малым и с большим шагом для такого расширения. Запрограммируйте для него интерпретатор.

Упражнение 3.14 (расширение: PCF со списками)**.** Дополним PCF следующими конструкциями: `nil` — обозначает пустой список; `cons n l` — обозначает список, первый элемент которого это натуральное число `n`, а `l` оставшаяся часть списка; **ifnil** `t` **then** `u` **else** `v` проверяет список на пустоту; `hd l` возвращает первый элемент списка `l`, а `tl l` — список `l` без первого элемента. Запишите семантические правила с малым и с большим шагом для такого расширения. Запрограммируйте для него интерпретатор. Реализуйте алгоритм сортировки таких списков.

Упражнение 3.15 (расширение: PCF с деревьями)**.** Дополним PCF следующими конструкциями: `L n` обозначает дерево, которое состоит из одного листа, помеченного натуральным числом `n`; `N t u` обозначает дерево с двумя поддеревьями, `t` и `u`; **ifleaf** `t` **then** `u` **else** `v` проверяет, имеет ли его первый аргумент вид `L n` или `N t u`; `content t` обозначает содержимое дерева `t`, если оно представляет из себя лист; `left t` и `right t` обозначают, соответственно, левое и правое поддерево дерева `t`, если оно не лист. Запишите семантические правила с малым и с большим шагом для такого расширения. Запрограммируйте для него интерпретатор.

Глава 4. Компиляция

Когда компьютер привозят с завода, он не способен интерпретировать термы PCF, равно как и термы Caml или Java. Чтобы такая возможность появилась, следует написать интерпретатор нужного языка на машинном языке, который понятен компьютеру. В прошлой главе были изложены основополагающие принципы интерпретации PCF и обсуждены некоторые детали реализации его интерпретатора на языках высокого уровня. Продолжая эту линию рассуждений, мы могли бы попытаться написать интерпретатор на машинном языке.

Другая возможность заключается в том, чтобы покинуть сферу интерпретации и бросить силы на разработку *компилятора*. Интерпретатор принимает на входе терм PCF и возвращает его значение. Компилятор, вместо того, представляет собой программу, которая получает терм PCF в качестве аргумента и создаёт программу на машинном языке, которая, будучи исполненной, выдаёт значение исходного терма. Другими словами, компилятор PCF это программа, транслирующая терм PCF в машинный код, то есть язык, который подразумевает непосредственное исполнение компьютером.

Одно из преимуществ использования компиляции состоит в том, что программа транслируется раз и навсегда во время компиляции, а не каждый раз при исполнении. С однократной компиляцией исполнение обычно происходит быстрее. Другое достоинство происходит из возможности компилятора компилировать самого себя — такой приём называется *раскруткой компилятора* (bootstrapping, см. упр. 4.4) — в то время как интерпретатор не может интерпретировать сам себя.

Реализация компилятора должна руководствоваться правилами операционной семантики (как и в случае интерпретатора). Для простоты сконцентрируемся на фрагменте PCF, где только функции могут быть определены рекурсивно, и воспользуемся операционной семантикой с большим шагом, содержащей рекурсивные замыкания (см. раздел 3.4).

Машинный язык, который будет использоваться, не применяется в промышленных масштабах, а предназначен для воображаемого компьютера. Такой тип компьютеров называется *абстрактными машинами*. Мы напишем программу, которая будет симулировать поведение этой машины. Использование абстрактной машины вызвано не только педагогическими

соображениями, существуют и практические мотивы: основные компиляторы для Caml и Java, к примеру, нацелены на абстрактные машины. Скомпилированные программы исполняются программой, которая симулирует работу абстрактной машины или транслирует их далее в машинный язык конкретного компьютера (что составляет уже вторую фазу компиляции).

4.1. Интерпретатор, написанный на языке без функций

В главе 2 была изложена операционная семантика с большим шагом для PCF, которая затем использовалась при создании интерпретатора. Например, правило

$$\frac{\mathtt{e} \vdash \mathtt{u} \hookrightarrow \mathtt{q} \qquad \mathtt{e} \vdash \mathtt{t} \hookrightarrow \mathtt{p}}{\mathtt{e} \vdash \mathtt{t + u} \hookrightarrow \mathtt{n}}, \text{ если } \mathtt{p + q = n}$$

переходит в следующий фрагмент кода интерпретатора PCF:

```
let rec interp env p = match p with
  | Plus(t,u) ->
    let w = interp env u
    in let v = interp env t
      in (match (v,w) with | (Const(n), Const(m)) ->
                                Const(n + m)
                           | ...)
  | ...
```

Поскольку Caml допускает локальные объявления, можно вычислить значение `interp env t` и восстановить сохранённое значение `w` после этого, даже если имя `w` связывалось с другими значениями в процессе этих вычислений.

Если попытаться написать интерпретатор на машинном языке или на любом языке без локальных объявлений, понадобится разработать механизм сохранения значения `w`, например, с использованием стека: можно интерпретировать терм `u`, положить результат на стек, затем интерпретировать терм `t`, а в конце снять верхнее значение со стека и сложить с результатом последней интерпретации.

В таком случае для интерпретации терма `((((1 + 2) + 3) + 4) + 5) + 6` придётся класть на стек число `6`, затем число `5`, ... затем число `2`, затем снять со стека верхнее значение (то есть `2`) и прибавить его к `1`, затем снять число `3` и прибавить его к предыдущему результату и т. д. В конце

концов нужно будет снять со стека число `6` и добавить его к предыдущему результату, чтобы получить окончательный результат: `21`.

4.2. От интерпретации к компиляции

Описанный интерпретатор может быть разделён на две программы. Первая выглядит как объект с двумя полями: поле, содержащее целое число, мы будем называть его *аккумулятор*, и поле со списком целых чисел, называемое *стеком*. Определим следующие операции:

— **Ldi** `n`: записывает `n` в аккумулятор,

— **Push**: кладёт аккумулятор на вершину стека,

— **Add**: складывает вершину стека со значением в аккумуляторе, записывает результат в аккумулятор и извлекает вершину стека.

Этот объект и есть наша абстрактная машина, а перечисленные инструкции составляют её машинный язык. Упомянутые поля называются *регистрами*.

Вторая программа принимает на вход терм PCF и выдаёт инструкции, которые будут последовательно исполнены на абстрактной машине. Если `t` это терм PCF, обозначим |`t`| список инструкций абстрактной машины, порождённых данной программой в процессе интерпретации терма. Например, для терма `((((1 + 2) + 3) + 4) + 5) + 6` будут выданы следующие инструкции:

> **Ldi** `6`, **Push**, **Ldi** `5`, **Push**, **Ldi** `4`, **Push**, **Ldi** `3`, **Push**, **Ldi** `2`, **Push**, **Ldi** `1`, **Add**, **Add**, **Add**, **Add**, **Add**.

Упражнение 4.1. Какие инструкции будут исполнены абстрактной машиной во время интерпретации терма `1 + (2 + (3 + (4 + (5 + 6))))`?

Такое разделение труда напоминает поведение водителя и пассажира в незнакомом городе: пассажир читает карту и даёт указания водителю, а тот следует указаниям, не имея представления, где вообще находится автомобиль.

Если бы пассажир мог давать инструкции просто глядя на карту, можно было бы записать их последовательность на компакт-диск, который водитель слушал бы в машине. В этом сценарии пассажиру необязательно находиться в автомобиле, чтобы направлять водителя. Аналогично интерпретатор мог бы записывать последовательность инструкций |`t`| в файл, который исполнялся бы позже на абстрактной машине. Именно так интерпретатор преобразуется в компилятор.

В общем случае считается, что абстрактная машина содержит в дополнение к аккумулятору и стеку третий регистр: *код*, список инструкций, которые необходимо исполнить. В самом начале абстрактная машина смотрит на первую инструкцию в регистре кода, исполняет её, затем переходит к следующей и т. д., пока не будет исчерпан весь список в регистре кода. Как мы увидим позже, тот факт, что исполнение одной инструкции может добавлять несколько новых в регистр кода, позволит создавать циклы и рекурсивные определения.

4.3. Абстрактная машина для PCF

4.3.1. Окружение

Пока что был скомпилирован лишь фрагмент PCF: числа и сложение. Можно ли распространить использованный принцип на весь язык?

Во-первых, напомним, что терм PCF должен исполняться в некотором окружении. В дополнение к регистрам аккумулятора, стека и кода абстрактной машине понадобится четвёртый регистр: *окружение*. Она также должна поддерживать инструкцию **Extend** `x`, которая добавляет в окружение определение `x = V`, где `V` это содержимое аккумулятора; понадобится также инструкция **Search** `x`, чтобы искать значение, связанное с `x`, в окружении и помещать его в аккумулятор.

Когда машина исполняет код, сгенерированный при компиляции нескольких вложенных применений, окружение меняется несколько раз, но в конце исходное окружение должно быть восстановлено. Таким образом, абстрактной машине понадобятся команды **Pushenv** и **Popenv**, чтобы сохранять окружение на стеке и доставать его оттуда. Эти операции обычно раскладываются на несколько элементарных команд работы со стеком, в которых участвуют элементы окружения, но мы таким разложением заниматься не будем.

4.3.2. Замыкания

Помимо чисел значениями в PCF являются также и замыкания. В дополнение к инструкции **Ldi** `n` понадобится инструкция **Mkclos** `(f, x, t)` с двумя переменными `f` и `x`, а также термом `t` в качестве аргументов. Эта инструкция создаёт замыкание $\langle$`f, x, t, e`$\rangle$, где `e` это содержимое регистра окружения, и сохраняет его в аккумуляторе.

4.3.3. Конструкции PCF

Не представляет сложности компиляция терма вида **fun** x → t или **fixfun** f x → t, поскольку можно просто сгенерировать инструкцию **Mkclos** (f, x, t) и получить замыкание, которое является подходящим значением термов этого вида.

Аналогично трудностей не вызовет терм x, для которого следует генерировать инструкцию **Search** x поиска в окружении значения, связанного с x.

Рассмотрим теперь компиляцию терма в форме t u. Соответствующее правило семантики с большим шагом выглядит так:

$$\frac{\begin{array}{c}\mathtt{e} \vdash \mathtt{u} \hookrightarrow \mathtt{W} \\ \mathtt{e} \vdash \mathtt{t} \hookrightarrow \langle \mathtt{f}, \mathtt{x}, \mathtt{t}', \mathtt{e}' \rangle \\ (\mathtt{e}', \mathtt{f} = \langle \mathtt{f}, \mathtt{x}, \mathtt{t}', \mathtt{e}' \rangle, \mathtt{x} = \mathtt{W}) \vdash \mathtt{t}' \hookrightarrow \mathtt{V}\end{array}}{\mathtt{e} \vdash \mathtt{t}\,\mathtt{u} \hookrightarrow \mathtt{V}}.$$

Чтобы интерпретировать терм t u в окружении e, начнём с интерпретации u в окружении e, в результате чего получится значение W. Затем интерпретируем t в окружении e и получим замыкание (f, x, t′, e′). Наконец, интерпретируем t′ в окружении (e′, f = ⟨f, x, t′, e′⟩, x = W), чтобы получить результат.

Посмотрим, как интерпретатор, запущенный в абстрактной машине, справится с данным термом: чтобы интерпретировать t u, машина сначала интерпретирует u, а результат кладёт на стек. Далее она интерпретирует t, получает замыкание ⟨f, x, t′, e′⟩ и помещает окружение e′, f = ⟨f, x, t′, e′⟩, x = W в регистр окружения, здесь W это значение на стеке, которое затем удаляется с вершины стека. Наконец, машина интерпретирует терм t′. Чтобы восстановить состояние окружения к концу всех действий, окружение кладётся на стек в самом начале и извлекается оттуда в конце.

Рассмотрим теперь компиляцию такого терма. Интерпретация терма u заменяется на исполнение списка инструкций |u|, аналогично интерпретации терма t соответствует исполнение набора инструкций |t|. Интерпретация t′ также переходит в исполнение последовательности инструкций |t′|. Однако здесь имеется трудность: t′ не является подтермом t u, он появляется из замыкания, полученного при интерпретации t. Нам нужно подправить понятие замыкания и заменить терм t в ⟨f, x, t, e⟩ некоторой последовательностью инструкций i. Таким образом, термы вида **fun** x → t и **fixfun** f x → t не должны компилироваться в **Mkclos** (f, x, t), вместо этого компиляция должна давать **Mkclos** (f, x, |t|), чтобы построить замыкание ⟨f, x, |t|, e⟩, где e является содержимым регистра окружения.

Наконец, нам понадобится включить в описание машины инструкцию **Apply**, которая принимает замыкание ⟨`f, x, i, e`⟩ из аккумулятора, помещает окружение `e, f = ⟨f, x, i, e⟩, x = W`, где `W` равен текущей вершине стека, в регистр окружения, удаляет вершину стека и добавляет в регистр кода последовательность инструкций `i`.

Терм `t u` можно скомпилировать в последовательность инструкций **Pushenv**, |`u`|, **Push**, |`t`|, **Apply**, **Popenv**.

Итак, наша абстрактная машина обладает набором инструкций **Ldi** `n`, **Push**, **Add**, **Extend** `x`, **Search** `x`, **Pushenv**, **Popenv**, **Mkclos** `(f, x, i)` и **Apply**. Последним штрихом станет добавление арифметических операций **Sub**, **Mult**, **Div** и проверки **Test** `(i, j)`, чтобы получить возможность компилировать операции `-`, `*`, `/` и **ifz**.

4.3.4. Использование индексов де Брауна

Чтобы упростить машину, можно использовать индексы де Брауна (см. раздел 3.3). Напомним, что инструкция **Search** `x` генерируется при компиляции переменных, а нам уже известно, что каждому вхождению переменной можно статически приписать некоторый индекс. То есть вместо **Search** `x` можно было бы использовать инструкцию **Search** `n`, где `n` является числом.

Индексы де Брауна могут быть вычислены прямо во время компиляции, достаточно лишь скомпилировать терм в окружении с переменными и скомпилировать переменную `x` в окружении `e` в инструкцию **Search** `n`, где `n` это позиция переменной `x` в окружении `e`, считая с конца.

Этот подход позволяет избавиться от переменных в окружениях, замыканиях и инструкциях **Mkclos** и **Extend**. Теперь абстрактная машина поддерживает инструкции **Ldi** `n`, **Push**, **Extend**, **Search** `n`, **Pushenv**, **Popenv**, **Mkclos** `i`, **Apply**, **Test** `(i, j)`, **Add**, **Sub**, **Mult** и **Div**.

4.3.5. Операционная семантика с малым шагом

Состояние машины, то есть содержимое её регистров, это кортеж, содержащий значение (аккумулятор), список, где каждый элемент является значением или списком значений (стек), список значений (окружение) и последовательность инструкций (код).

Малый шаг исполнения включает получение инструкции из регистра кода и её исполнение. Нетрудно определить машинную семантику с малым шагом:

— `(a, s, e, ((`**Mkclos** `i), c))` $\longrightarrow$ `(⟨i, e⟩, s, e, c)`,

— `(a, s, e, (`**Push**`, c)) ⟶ (a, (a, s), e, c)`,

— `(a, s, e, (`**Extend**`, c)) ⟶ (a, s, (e, a), c)`,

— `(a, s, e, ((`**Search** `n), c)) ⟶ (V, s, e, c)`, где `V` это `n`-ое значение в окружении `e` (считая с конца),

— `(a, s, e, (`**Pushenv**`, c)) ⟶ (a, (e, s), e, c)`,

— `(a, (e′, s), e, (`**Popenv**`, c)) ⟶ (a, s, e′, c)`,

— `(⟨i, e′⟩, (W, s), e, (`**Apply**`, c)) ⟶ (⟨i, e′⟩, s, (e′, ⟨i, e′⟩, W), i c)`,

— `(a, s, e, ((`**Ldi** `n), c)) ⟶ (n, s, e, c)`,

— `(n, (m, s), e, (`**Add**`, c)) ⟶ (n + m, s, e, c)`,

— `(n, (m, s), e, (`**Sub**`, c)) ⟶ (n - m, s, e, c)`,

— `(n, (m, s), e, (`**Mult**`, c)) ⟶ (n * m, s, e, c)`,

— `(n, (m, s), e, (`**Div**`, c)) ⟶ (n / m, s, e, c)`,

— `(0, s, e, ((`**Test** `(i, j)), c)) ⟶ (0, s, e, i c)`,

— `(n, s, e, ((`**Test** `(i, j)), c)) ⟶ (n, s, e, j c)`, где число `n` отлично от нуля.

Нередуцируемый терм это кортеж, где четвёртая компонента — содержимое регистра кода — пуста. Если `i` это последовательность инструкций, а терм `(0, [], [], i)` редуцируется к нередуцируемому терму вида `(V, _, _, [])`, то говорят, что `V` это результат исполнения `i`, и пишут `i ⇒ V`.

4.4. Компиляция PCF

Теперь мы готовы записать правила компиляции PCF:

— $|\mathtt{x}|_{\mathtt{e}}$ = **Search** `n`, где `n` это позиция `x` в окружении `e`;

— $|\mathtt{t\ u}|_{\mathtt{e}}$ = **Pushenv**, $|\mathtt{u}|_{\mathtt{e}}$, **Push**, $|\mathtt{t}|_{\mathtt{e}}$, **Apply**, **Popenv**;

— $|$**fun** `x` $\to \mathtt{t}|_{\mathtt{e}}$ = **Mkclos** $|\mathtt{t}|_{\mathtt{e},\_,\mathtt{x}}$;

— $|$**fixfun** `f x` $\to \mathtt{t}|_{\mathtt{e}}$ = **Mkclos** $|\mathtt{t}|_{\mathtt{e},\mathtt{f},\mathtt{x}}$;

— $|\mathtt{n}|_{\mathtt{e}}$ = **Ldi** `n`;

— $|\mathtt{t + u}|_{\mathtt{e}} = |\mathtt{u}|_{\mathtt{e}}$, **Push**, $|\mathtt{t}|_{\mathtt{e}}$, **Add**;

— $|\mathtt{t - u}|_{\mathtt{e}} = |\mathtt{u}|_{\mathtt{e}}$, **Push**, $|\mathtt{t}|_{\mathtt{e}}$, **Sub**;

— $|\mathtt{t * u}|_{\mathtt{e}} = |\mathtt{u}|_{\mathtt{e}}$, **Push**, $|\mathtt{t}|_{\mathtt{e}}$, **Mult**;

— $|\texttt{t / u}|_{\texttt{e}} = |\texttt{u}|_{\texttt{e}}$, **Push**, $|\texttt{t}|_{\texttt{e}}$, **Div**;

— $|$**ifz** `t` **then** `u` **else** `v`$|_{\texttt{e}} =$ $|\texttt{t}|_{\texttt{e}}$, **Test** `(`$|\texttt{u}|_{\texttt{e}}$, $|\texttt{v}|_{\texttt{e}}$`)`;

— $|$**let** `x = t in u`$|_{\texttt{e}} =$ **Pushenv**, $|\texttt{t}|_{\texttt{e}}$, **Extend**, $|\texttt{u}|_{\texttt{e,x}}$, **Popenv**.

Например, компиляция терма

let `f =` **fixfun** `f x` $\to$ `(`**ifz** `x` **then** `1` **else** `(x*(f (x - 1)))` **in** `f 6`

даёт набор инструкций:

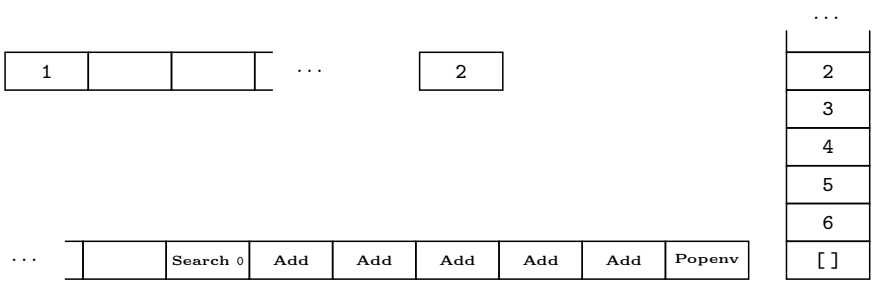
Pushenv, **Mkclos** [**Search** `0`, **Test** `([Ldi 1],`
`[`**Pushenv**, **Ldi** `1`, **Push**, **Search** `0`, **Sub**, **Push**,
Search `1`, **Apply**, **Popenv**, **Push**, **Search** `0`, **Mult**`])`],
Extend, **Pushenv**, **Ldi** `6`, **Push**, **Search** `0`,
Apply, **Popenv**, **Popenv**,

в результате исполнения которых получается `720`.

Корректность компиляции и семантики абстрактной машины могут быть сформулированы так: если `V` это число, то $\vdash$ `t` $\hookrightarrow$ `V` в том и только том случае, когда $|\texttt{t}| \Rightarrow \texttt{V}$.

Упражнение 4.2. Реализуйте абстрактную машину и компилятор для языка PCF.

Продемонстрируем состояние абстрактной машины к началу 14-го шага выполнения программы **Pushenv**, **Ldi** `1`, **Extend**, **Ldi** `6`, **Push**, **Ldi** `5`, **Push**, **Ldi** `4`, **Push**, **Ldi** `3`, **Push**, **Ldi** `2`, **Push**, **Search** `0`, **Add**, **Add**, **Add**, **Add**, **Add**, **Popenv**:



Упражнение 4.3. Расширим PCF операциями с деревьями, описанными в упр. 3.15. Реализуйте абстрактную машину и компилятор для такого расширения PCF.

Упражнение 4.4 (раскрутка компилятора)**.** Многие структуры данных могут быть реализованы на основе деревьев, описанных в упр. 3.15. Для начала представим натуральное число `n` как `L n`. Символ `c` может быть представлен деревом `L n`, где `n` обозначает некоторый код, например ASCII-код символа `c`. Если $\mathtt{t_1}, \mathtt{t_2}, \ldots, \mathtt{t_n}$ это деревья, то список $\mathtt{t_1}, \mathtt{t_2}, \ldots, \mathtt{t_n}$ можно записать как дерево $\mathtt{N(t_1, N(t_2, \ldots, N(t_n, L\,0)\ldots))}$. Наконец, если значение типа задано конструктором, который сам является представимым, то такое значение можно задать с помощью нумерации конструкторов следующим образом: значение $\mathtt{C(V_1, V_2, \ldots, V_n)}$ представляется списком $\mathtt{L\,p, t_1, t_2, \ldots, t_n}$, где `p` это число, ассоциированное с конструктором `C`, а $\mathtt{t_1, t_2, \ldots, t_n}$ представляют значения $\mathtt{V_1, V_2, \ldots, V_n}$.

Можно было бы, в частности, представить таким образом программы, написанные на расширенном PCF или на языке абстрактной машины из упр. 4.3. Модифицируйте компилятор и абстрактную машину из упр. 4.3 так, чтобы они могли работать с программами, представленными в виде бинарных деревьев. Абстрактная машина будет иметь два входа: скомпилированная программа, представленная в виде дерева, и некоторое значение; машина будет применять программу к значению.

Транслируйте компилятор из упр. 4.3 в PCF. После написания такого компилятора на PCF скомпилируйте его компилятором, написанным при выполнении упр. 4.3. В результате получится первый компилятор, работающий на абстрактной машине PCF. Скомпилируйте этот компилятор самим собой. Убедитесь, что он генерирует тот же код, что и компилятор, созданный в упр. 4.3. Если всё верно, мы можем уничтожить первый компилятор и пользоваться вторым: этот приём называется раскруткой компилятора.

Глава 5. PCF с типами

Во второй главе мы заметили, что в отличие от математических функций область определения функций PCF не задаётся. Это делает возможным применение функции **fun** `x` $\to$ `x + 1` к функции **fun** `x` $\to$ `x`, даже если такое применение не имеет никакого смысла.

Иногда бывает удобным иметь возможность применить произвольный объект к любому другому. Например, вполне можно применить тождественную функцию **fun** `x` $\to$ `x` к самой себе, получив при этом терм `(`**fun** `x` $\to$ `x)(`**fun** `x` $\to$ `x)`, редуцируемый к **fun** `x` $\to$ `x`. Вообще говоря, тождественная функция в PCF определена для любого объекта, тогда как в математике её всегда нужно ограничивать некоторой областью определения. Сама возможность применения объекта к самому себе оказалась ключевой при демонстрации того, как с помощью применения и **fun** можно эмулировать конструкцию **fix** (см. упр. 2.10).

В то же время неограниченное применение одного объекта к другому способно привести к целому ряду проблем. К примеру, мы наблюдали, что термы `1 2`, `1 + (`**fun** `x` $\to$ `x)`, **ifz fun** `x` $\to$ `x` **then** `1` **else** `2` оказывались в операционной семантике для PCF с малым шагом нередуцируемыми замкнутыми термами и не являлись при этом значениями.

Операционная семантика с большим шагом вообще не приписывает никакого значения термам вида `(`**fun** `x` $\to$ `x) 1 2`. На практике при попытке интерпретировать терм вида `tu`, где `t` сводится к числу, а не к ожидаемому терму вида **fun**, генерируется ошибка. Такая ошибка будет обнаружена только во время исполнения, а вовсе не статически (то есть до исполнения), как того бы хотелось.

Тот факт, что для функций PCF не указывается область определения, затрудняет также построение денотационной семантики PCF.

Целью этой главы является определение некоторого варианта PCF, в котором функциям приписывается соответствующая область определения, а также доказательство того, что интерпретация любой корректно заданной на этом языке функции не может привести к возникновению обсуждавшихся выше ошибок. Также будет представлена простая денотационная семантика этого языка.

5.1. Типы

В математике областью определения функции является некоторое (произвольное) множество. Например, мы можем определить функцию `m`, действующую из $2\mathbb{N}$ в $\mathbb{N}$, которая ставит в соответствие каждому чётному числу его половину. Тогда для проверки, является ли, скажем, выражение `m (3 + (4 + 1))` корректным, то есть принадлежит ли аргумент области определения функции, необходимо выяснить, чётно ли число `3 + (4 + 1)`. Вообще говоря, для произвольных множеств задача проверки принадлежности элемента множеству является неразрешимой. Следовательно, в общем случае неразрешима и задача проверки корректности термов. К тому же, чтобы узнать, возникнет ли ошибка при исполнении такого терма, как **ifz** `t` **then** `u` **else** `v`, необходимо выяснить, является ли значение `t` натуральным числом или же термом вида **fun** (в этом случае говорить о чётности не приходится).

Два этих замечания приводят нас к необходимости ограничения класса тех множеств, которые могут использоваться для задания областей определения функций. Назовём множества из соответствующего класса *типами*.

5.1.1. PCF с типами

Типы в PCF определяются индуктивно:

— `nat` — то есть $\mathbb{N}$ — это тип;

— если `A` и `B` типы, то `A` $\to$ `B`, а именно множество всех функций из `A` в `B` — также тип.

Теперь можно определять типы с помощью языка, включающего константу `nat` и символ $\to$ с двумя аргументами, не связывающий никакие переменные. Такой терм также называется *типом*.

Функции PCF записывались как **fun** `x` $\to$ `t`, но теперь придётся указывать тип переменной `x`. Поэтому, к примеру, будем писать **fun** `x:nat` $\to$ `x` для тождественной функции, действующей на множестве натуральных чисел, и **fun** `x : (nat` $\to$ `nat)` $\to$ `x` для тождественной функции, действующей на множестве функций из натуральных чисел в натуральные числа. В общем, символ **fun** теперь будет иметь два аргумента: тип и терм; он будет связывать переменную во втором аргументе. Типизированный вариант языка PCF это язык с двумя сортами объектов: термами и типами, арностью символа **fun** является `((type), (term, term), term)`. В символах **fix** и **let** также необходимо указывать тип связанной переменной.

Подытожим, что именно содержится в типизированном PCF:

— термовый символ **fun** с типовым и термовым аргументами, связывающий переменную во втором аргументе;

— термовый символ α с двумя термовыми аргументами, не связывающий никакие переменные;

— бесконечное количество термовых констант, представляющих натуральные числа;

— четыре термовых символа `+`, `-`, `*` и `/`, каждый с двумя аргументами, не связывающие никакие переменные;

— термовый символ **ifz** с тремя аргументами, не связывающий никакие переменные;

— термовый символ **fix** с одним типовым и одним термовым аргументами, связывающий переменную во втором аргументе;

— термовый символ **let** с тремя аргументами, первый из которых типовый, а остальные термовые, связывающий переменную в третьем аргументе;

— типовая константа `nat`;

— типовый символ $\to$, с двумя типовыми аргументами, не связывающий никакие переменные.

Мы также можем определить синтаксис типизированного варианта PCF индуктивно:

`A` = `X`

| `nat`

| `A` $\to$ `A`

`t` = `x`

| **fun** `x:A` $\to$ `t`

| `t t`

| `n`

| `t + t` | `t - t` | `t * t` | `t / t`

| **ifz** `t` **then** `t` **else** `t`

| **fix** `x:A t`

| **let** `x:A = t` **in** `t`

5.1.2. Отношение типизации

Теперь с помощью индукции можно определить отношение `t:A` (читается как «терм `t` имеет тип `A`»). Более строго, с помощью индукции мы будем определять тернарное отношение $\mathtt{e} \vdash \mathtt{t} : \mathtt{A}$, как это уже делалось для отношения интерпретации, где `t` это терм, возможно, со свободными переменными, а `e` это типовое окружение, ассоциирующее некоторый тип с каждой из переменных. Это индуктивное определение похоже на индуктивное определение операционной семантики PCF с большим шагом. Можно представлять, что это операционная семантика для языка с тем же синтаксисом, что и PCF, интерпретация терма в котором возвращает не значение, а тип — по этой причине соответствующий процесс называется *абстрактной интерпретацией* терма.

$$\frac{}{\mathtt{e} \vdash \mathtt{x{:}A}} \text{ если } \mathtt{e} \text{ содержит } \mathtt{x{:}A},$$

$$\frac{\mathtt{e} \vdash \mathtt{u{:}A} \qquad \mathtt{e} \vdash \mathtt{t{:}A \to B}}{\mathtt{e} \vdash \mathtt{t\ u{:}B}},$$

$$\frac{(\mathtt{e}, \mathtt{x{:}A}) \vdash \mathtt{t{:}B}}{\mathtt{e} \vdash \mathbf{fun}\ \mathtt{x{:}A \to t{:}A \to B}},$$

$$\frac{}{\mathtt{e} \vdash \mathtt{n{:}nat}},$$

$$\frac{\mathtt{e} \vdash \mathtt{u{:}nat} \qquad \mathtt{e} \vdash \mathtt{t{:}nat}}{\mathtt{e} \vdash \mathtt{t} \otimes \mathtt{u{:}nat}},$$

$$\frac{\mathtt{e} \vdash \mathtt{t{:}nat} \qquad \mathtt{e} \vdash \mathtt{u{:}A} \qquad \mathtt{e} \vdash \mathtt{v{:}A}}{\mathtt{e} \vdash \mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v{:}A}},$$

$$\frac{(\mathtt{e}, \mathtt{x{:}A}) \vdash \mathtt{t{:}A}}{\mathtt{e} \vdash \mathbf{fix}\ \mathtt{x{:}A\ t{:}A}},$$

$$\frac{\mathtt{e} \vdash \mathtt{t{:}A} \qquad (\mathtt{e}, \mathtt{x{:}A}) \vdash \mathtt{u{:}B}}{\mathtt{e} \vdash \mathbf{let}\ \mathtt{x{:}A = t}\ \mathbf{in}\ \mathtt{u{:}B}}.$$

В первом правиле во внимание принимается только самое правое объявление `x`, все остальные скрыты.

Язык включает переменные различных сортов, в частности, типовые переменные, для которых мы будем использовать заглавные буквы. По-

скольку нет символов, связывающих типовые переменные, замкнутые термы содержать типовые переменные не могут. Более того, если замкнутый терм `t` имеет тип `A` в пустом окружении, то тип `A` также должен быть замкнут. Поэтому здесь типовые переменные практически не используются, они пригодятся только в следующей главе.

Предположим, что `e` это окружение, а `t` это терм. Рассуждая индукцией по `t`, мы можем показать, что терм `t` имеет в окружении `e` не более одного типа.

Пользуясь приведёнными выше правилами, можно построить алгоритм *проверки типов*. Алгоритм будет проверять, имеет ли терм `t` тип в окружении `e`, и если имеет, возвращать этот тип в качестве результата. Это достигается рекурсивной типизацией непосредственных подтермов данного терма и вычислением типа терма по типам подтермов.

Упражнение 5.1. Запрограммируйте алгоритм проверки типов для языка PCF.

Отношение редукции на языке PCF с типами по-прежнему обладает свойством слияния, к тому же типы привносят дополнительное свойство: все термы, не содержащие конструкцию **fix**, завершаются — это утверждение носит название теоремы Тейта (Tait). В типизированном PCF невозможно построить такой терм, как `(`**fun** `x` $\to$ `(x x))(`**fun** `x` $\to$ `(x x))`, не завершающийся даже в отсутствие **fix**.

Упражнение 5.2. Напишите правила типизации для варианта PCF с использованием индексов де Брауна вместо имён переменных (см. раздел 3.3).

Упражнение 5.3. Добавим в язык PCF пары, расширив его конструкциями, описанными в упр. 3.13, и введём символ $\times$, обозначающий декартово произведение двух типов. Напишите правила типизации и реализуйте проверку типов для этого расширения PCF.

Упражнение 5.4. Добавим в язык PCF списки, расширив его конструкциями, описанными в упр. 3.14, и введём для списков тип `natlist`. Напишите правила типизации и реализуйте проверку типов для этого расширения PCF.

Упражнение 5.5. Добавим в язык PCF деревья, расширив его конструкциями, описанными в упр. 3.15, и введём для деревьев тип `nattree`. Напишите правила типизации и реализуйте проверку типов для этого расширения PCF.

5.2. Отсутствие ошибок во время выполнения

Покажем теперь, что интерпретация корректно типизированного терма не может привести к ошибке с типами во время выполнения. Для этого можно использовать операционную семантику как с малым шагом, так и с большим, доказательство, правда, будет слегка различаться.

5.2.1. Использование операционной семантики с малым шагом

При использовании операционной семантики с малым шагом требуемое свойство может быть сформулировано следующим образом: результат вычисления типизированного замкнутого терма, если он существует, является значением. Иными словами, типизированный замкнутый терм вычисляется к натуральному числу или замкнутому терму вида **fun** $\mathtt{x} \to \mathtt{t}$; он никогда не окажется тупиковым: $\mathtt{V_1 V_2}$, где $\mathtt{V_1}$ и $\mathtt{V_2}$ нередуцируемые замкнутые термы, причём $\mathtt{V_1}$ не терм вида **fun** $\mathtt{x} \to \mathtt{t}$, $\mathtt{V_1} \otimes \mathtt{V_2}$, где $\mathtt{V_1}$ и $\mathtt{V_2}$ нередуцируемые замкнутые термы, не являющиеся числами, или **ifz** $\mathtt{V_1}$ **then** $\mathtt{V_2}$ **else** $\mathtt{V_3}$, где $\mathtt{V_1}$, $\mathtt{V_2}$ и $\mathtt{V_3}$ нередуцируемые замкнутые термы, причём $\mathtt{V_1}$ не является числом.

Первая лемма, которую мы не будем здесь доказывать, обычно называется редукцией субъекта. Она утверждает, что если замкнутый терм `t` типа `A` сводится за один шаг к терму `u` ($\mathtt{t} \rhd \mathtt{u}$), то `u` также имеет тип `A`. Отсюда можно вывести, что если замкнутый терм `t` типа `A` сводится за произвольное число шагов к терму `u` ($\mathtt{t} \rhd^* \mathtt{u}$), то `u` также имеет тип `A`.

На следующем шаге доказательства необходимо показать, что терм вида **fun** не может иметь тип `nat`, точно так же числовая константа не может иметь тип вида $\mathtt{A} \to \mathtt{B}$. Это делается структурной индукцией по отношению типизации.

Доказательство продолжается обоснованием того факта, что нередуцируемый замкнутый терм `t` типа `nat` является константой, представляющей натуральное число, тогда как такой же терм типа $\mathtt{A} \to \mathtt{B}$ имеет вид **fun**. Для этого необходимо провести структурную индукцию по терму `t`.

Так как `t` является замкнутым термом, он не может быть переменной. В силу нередуцируемости он не может быть ни **fix**, ни **let**.

Покажем, что `t` не может оказаться применением, арифметической или условной операцией. Если `t` это применение $\mathtt{t} = \mathtt{u}\ \mathtt{v}$, то `u` имеет тип вида $\mathtt{C} \to \mathtt{D}$. Согласно предположению индукции этот терм должен иметь вид **fun**, а значит, `t` является редексом, что противоречит предположению (`t` нередуцируем). Если `t` это арифметическая операция $\mathtt{t} = \mathtt{u} \otimes \mathtt{v}$, то `u` и `v` типа `nat`. По предположению индукции они являются числовыми константами, и, следовательно, `t` снова редекс, то есть мы опять пришли к

противоречию. Наконец, если `t` это терм вида **ifz** `u` **then** `v` **else** `w`, то `u` — числовая константа, а `t` — редекс, что вновь приводит к противоречию с нередуцируемостью `t`.

Таким образом, нередуцируемый замкнутый терм `t` является либо числовой константой, либо термом вида **fun**. Если его тип `nat`, то это константа, если же $\texttt{A} \to \texttt{B}$, то **fun**.

Если корректно типизированный замкнутый терм можно привести к нередуцируемому замкнутому терму, то этот нередуцируемый терм также будет корректно типизирован, а значит, окажется либо числовой константой, либо термом вида **fun**.

5.2.2. Использование операционной семантики с большим шагом

Операционная семантика с большим шагом требует иной формулировки рассматриваемого свойства. Причина в том, что в семантике такого стиля с термами (даже если они неправильно типизированы) могут быть ассоциированы только значения. Можно было бы сказать, что правила операционной семантики с большим шагом неполны, поскольку они не предписывают, как следует связывать значение с применением, значение левой части которого является числовой константой, или, другой пример, как определить значение арифметической операции, один из аргументов которой имеет вид **fun**, или же, наконец, какое значение можно сопоставить условной операции, первый аргумент которой имеет вид **fun**. Тем не менее, для корректно типизированных термов правила полны. Другими словами, три только что приведённых примера просто не могут возникнуть.

Начнём с леммы о *сохранении типа при интерпретации*, утверждающей, что если замкнутый терм `t` имеет тип `A`, то его значение, если оно существует, также имеет тип `A`. Эта лемма является аналогом леммы о редукции субъекта в случае операционной семантики с малым шагом.

Затем покажем, что, как и для семантики с малым шагом, терм вида **fun** не может иметь тип `nat`, а числовая константа не может иметь тип вида $\texttt{A} \to \texttt{B}$.

Поскольку известно, что значением терма является либо число, либо терм вида **fun**, можно заключить, что значением терма типа `nat` является числовая константа, а значением терма типа $\texttt{A} \to \texttt{B}$ является терм вида **fun**. Следовательно, при интерпретации корректно типизированного терма левая компонента применения всегда будет интерпретироваться как терм вида **fun**, а аргументы арифметических операций и первый аргумент **ifz** — как числовые константы.

Упражнение 5.6 (эквивалентность семантик)**.** Покажите, что вычисление корректно типизированного терма приводит к некоторому результату в операционной семантике с малым шагом и вызовом по имени в том и только в том случае, когда вычисление того же терма приводит к результату в операционной семантике с большим шагом. Более того, результаты в обоих случаях совпадают. Покажите также, что аналогичное свойство имеет место и при вызове по значению.

Проверьте, имеет ли место это утверждение для бестипового варианта PCF? Подсказка: каков результат вычисления терма `((`**fun** `x` $\to$ `x) 1) 2`?

5.3. Денотационная семантика для PCF с типами

5.3.1. Тривиальная семантика

Выше мы уже замечали, что одной из целей функциональных языков является сокращение расстояния между понятиями программы и функции. Другими словами, цель в приближении программы к её денотационной семантике.

Мы также упоминали, что для языка PCF без типов дать денотационную семантику довольно трудно, поскольку функции в этом случае не обладают областями определения. Сейчас, когда система типов для PCF уже есть, описать денотационную семантику гораздо легче.

Свяжем с каждым типом множество:

— $[\![\mathtt{nat}]\!] = \mathbb{N}$,

— $[\![\mathtt{A} \to \mathtt{B}]\!] = [\![\mathtt{A}]\!] \to [\![\mathtt{B}]\!]$,

а с каждым термом `t` типа `A` элемент $[\![\mathtt{t}]\!]$ множества $[\![\mathtt{A}]\!]$. Если в терме `t` имеются свободные переменные, связанные с ними значения будут определяться посредством *семантического окружения* `e`.

— $[\![\mathtt{x}]\!]_\mathtt{e} = \mathtt{a}$, если `e` содержит пару `x = a`;

— $[\![\textbf{fun}\ \mathtt{x{:}A} \to \mathtt{t}]\!]_\mathtt{e} = \textbf{fun}\ \mathtt{a{:}}[\![\mathtt{A}]\!] \to [\![\mathtt{t}]\!]_{\mathtt{e},\mathtt{x=a}}$;

— $[\![\mathtt{t\ u}]\!]_\mathtt{e} = [\![\mathtt{t}]\!]_\mathtt{e}[\![\mathtt{u}]\!]_\mathtt{e}$;

— $[\![\mathtt{n}]\!]_\mathtt{e} = \mathtt{n}$;

— $[\![\mathtt{t + u}]\!]_\mathtt{e} = [\![\mathtt{t}]\!]_\mathtt{e} + [\![\mathtt{u}]\!]_\mathtt{e}$, $[\![\mathtt{t - u}]\!]_\mathtt{e} = [\![\mathtt{t}]\!]_\mathtt{e} - [\![\mathtt{u}]\!]_\mathtt{e}$,
$[\![\mathtt{t * u}]\!]_\mathtt{e} = [\![\mathtt{t}]\!]_\mathtt{e} * [\![\mathtt{u}]\!]_\mathtt{e}$, $[\![\mathtt{t\ /\ u}]\!]_\mathtt{e} = [\![\mathtt{t}]\!]_\mathtt{e}\ /\ [\![\mathtt{u}]\!]_\mathtt{e}$;

— $[\![\textbf{ifz}\ \mathtt{t}\ \textbf{then}\ \mathtt{u}\ \textbf{else}\ \mathtt{v}]\!]_\mathtt{e} = [\![\mathtt{u}]\!]_\mathtt{e}$, если $[\![\mathtt{t}]\!]_\mathtt{e} = \mathtt{0}$ и $[\![\mathtt{v}]\!]_\mathtt{e}$ в противном случае;

— $[\![\textbf{let}\ \mathtt{x{:}A = t}\ \textbf{in}\ \mathtt{u}]\!]_\mathtt{e} = [\![\mathtt{u}]\!]_{\mathtt{e},\mathtt{x}=[\![\mathtt{t}]\!]_\mathtt{e}}$.

Это действительно тривиально: программа это функция, а её семантика это та же функция. Достижение подобной «тривиальности» и является одной из целей проектирования функциональных языков.

Следует отметить два факта. Во-первых, деление на `0` приводит к ошибке PCF, поскольку оно не определено в математике. Точнее говоря, мы должны добавить значение `error` к каждому множеству $[\![\mathtt{A}]\!]$, подправив при этом данное выше определение. Во-вторых, мы совершенно позабыли о конструкции **fix**.

5.3.2. Завершаемость

Единственной конструкцией с нетривиальной семантикой оказывается **fix**. Причина в том, что она далеко не каждый день встречается в математике. В отличие от PCF, математические определения могут использовать неподвижные точки только тех функций, которые их действительно имеют, если же имеется несколько неподвижных точек, то важно явно указать, какая именно из них берётся. При определении PCF мы оставили эти вопросы в стороне, теперь самое время к ним вернуться.

Возьмём для примера функцию **fun** `x : nat` $\to$ `x + 1`, не имеющую неподвижной точки. В PCF допустимо построение терма **fix** `x:nat (x + 1)`. Точно так же неподвижной точки не имеет функция **fun** `f:(nat` $\to$ `nat)` $\to$ **fun** `x:nat` $\to$ `(f x) + 1`, однако терм **fix** `f:(nat` $\to$ `nat)` **fun** `x:nat` $\to$ `(f x) + 1` построить можно. С другой стороны, у функции **fun** `x : nat` $\to$ `x` много неподвижных точек, а терм **fix** `x:nat x` по-прежнему строится.

Когда определялась операционная семантика PCF, мы вводили следующее правило редукции:

$$\textbf{fix}\ \mathtt{x{:}A\ t} \longrightarrow (\textbf{fix}\ \mathtt{x{:}A\ t/x})\ \mathtt{t},$$

объясняющее идею неподвижной точки. Используя это правило, можно редуцировать терм `a = fix x:nat (x + 1)` до `a + 1`, затем до `(a + 1) + 1` и т. д., не достигая никогда нередуцируемого терма. Похожим образом, если `g = fix f:(nat` $\to$ `nat) fun x:nat` $\to$ `(f x) + 1`, то терм `g 0` можно за два шага свести к `(g0) + 1`, затем к `((g0) + 1) + 1` и т. д., снова никогда не завершаясь. В точности то же самое происходит с термом `b = fix x : nat x`, который редуцируется к `b`, затем к `b` и т. д. Иными словами, в PCF может случиться так, что при взятии неподвижной точки функции, которая такую точку не имеет или же имеет более одной, выполнение программы не завершается.

Похожим образом обстоит дело в Caml, где программа

```
let rec f x = (f x) + 1 in (f 0)
```

зацикливается, или в Java, где аналогично ведёт себя программа

```
class Loop {
    static int f (int x) {return f(x) + 1;}
    static public void main (String [ ] args) {
        System.out.println(f(0));
    }
}
```

Попадаются даже такие функции, например, **fun** `x : nat` $\to$ `x + x`, которые, имея единственную неподвижную точку, тем не менее в результате применения к ним конструкции **fix** приводят к незавершающимся вычислениям: **fix** `x:nat (x + x)`.

В общем, чтобы понять денотационную семантику оператора неподвижной точки, необходимо сначала понять семантику незавершающихся термов.

Операционная семантика с малым шагом не связывает с такими термами никакого результата: не существует такого терма `V`, для которого **fix** `x : nat (x + 1)` $\hookrightarrow$ `V`. Точно так же операционная семантика с большим шагом не прибавляет информации. Как уже говорилось, отношение $\hookrightarrow$ можно пополнить значением $\bot$, таким что **fix** `x:nat (x + 1)` $\hookrightarrow \bot$.

Те же возможности имеются и в случае денотационной семантики. Можно либо определить частичную функцию $[\![\ ]\!]$, оставив при этом значение $[\![$**fix** `x : nat (x + 1)`$]\!]$ неопределённым, либо добавить к $[\![$`nat`$]\!]$ новое значение $\bot$ и положить $[\![$**fix** `x:nat (x + 1)`$]\!] = \bot$.

Если мы добавим значение $\bot$, то интерпретация терма вида `t+u` будет получаться из интерпретации `u` и `t`, а если один из этих термов не завершается, то не завершается и вычисление `t + u` в целом. Таким образом, денотационную семантику терма вида `t+u` можно определить следующим образом:

— $[\![$`t + u`$]\!] = [\![$`t`$]\!] + [\![$`u`$]\!]$, если $[\![$`t`$]\!]$ и $[\![$`u`$]\!]$ натуральные числа;

— $[\![$`t + u`$]\!] = \bot$, если $[\![$`t`$]\!] = \bot$ или $[\![$`u`$]\!] = \bot$.

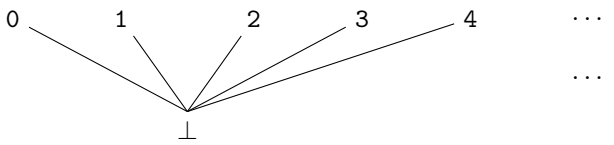
Отметим, что функция $[\![$**fun** `x : nat` $\to$ `x + 1`$]\!]$, не имевшая неподвижной точки, пока во множестве значений не было $\bot$, обрела её: это как раз $\bot$. Именно это значение определит семантику незавершающегося терма **fix** `x:nat (x + 1)`. Функция $[\![$**fun** `x:nat` $\to$ `x`$]\!]$ с несколькими неподвижными точками получает теперь ещё одну, $\bot$, и именно её мы выбираем в качестве семантики незавершающегося терма **fix** `x:nat (x + x)`. Функция

$[\![\textbf{fun}\ \texttt{x : nat} \to \texttt{x + x}]\!]$, имевшая единственную неподвижную точку `0`, теперь получает две: `0` и $\bot$, и снова в качестве семантики незавершающегося терма **fix** `x:nat (x + x)` мы избираем $\bot$.

Все упоминавшиеся ранее функции имеют теперь неподвижные точки, причём, если их несколько, в том числе $\bot$, то именно её мы выбираем в качестве искомого значения.

5.3.3. Отношение порядка Скотта

Уточним обсуждавшиеся выше идеи, определив отношение порядка на множестве $[\![\texttt{nat}]\!]$, называемое *отношением порядка Скотта*, следующим образом:



и определим $[\![\textbf{fix}\ \texttt{x:nat t}]\!]$ как наименьшую неподвижную точку функции $\textbf{fun}\ \texttt{x:nat} \to \texttt{t}$, что приведёт к использованию неподвижной точки $\bot$ всюду, где имеется более одной неподвижной точки. Остаётся доказать, что наименьшая неподвижная точка существует, для этого мы воспользуемся теоремой о неподвижной точке. Чтобы применить эту теорему, нужно показать, что определённое на $[\![\texttt{nat}]\!]$ отношение порядка слабо полно, а семантика программы типа $\texttt{nat} \to \texttt{nat}$ является непрерывной функцией.

Вообще говоря, для каждого типа `A` мы будем строить множество $[\![\texttt{A}]\!]$, снабжённое отношением слабо полного порядка, а затем покажем, что семантика программы типа $\texttt{A} \to \texttt{B}$ является непрерывной функцией из $[\![\texttt{A}]\!]$ в $[\![\texttt{B}]\!]$.

Начнём с определения множеств $[\![\texttt{A}]\!]$. Множество `nat` определяется как $\mathbb{N} \cup \{\bot\}$ с показанным выше отношением порядка. Множество $[\![\texttt{A} \to \texttt{B}]\!]$ определяется как множество всех непрерывных функций из $[\![\texttt{A}]\!]$ в $[\![\texttt{B}]\!]$, отношение порядка на котором вводится следующим образом: $\texttt{f} \leqslant \texttt{g}$, если для всех `x` из $[\![\texttt{A}]\!]$ выполняется $\texttt{f x} \leqslant \texttt{g x}$.

Можно показать, что эти отношения порядка слабо полны. Порядок на $[\![\texttt{nat}]\!]$ слабо полный, поскольку любая возрастающая последовательность является либо постоянной, либо имеет вид $\bot, \bot, \ldots, \bot, \texttt{n}, \texttt{n}, \ldots$, а значит, в обоих случаях имеет предел.

Покажем теперь, что если отношения порядка на $[\![\texttt{A}]\!]$ и $[\![\texttt{B}]\!]$ слабо полны, то таким же будет порядок на $[\![\texttt{A} \to \texttt{B}]\!]$. Рассмотрим возрастающую

последовательность функций $\mathtt{f_n}$ над $[\![\mathtt{A} \to \mathtt{B}]\!]$. Пользуясь определением порядка на $[\![\mathtt{A} \to \mathtt{B}]\!]$, для всех $\mathtt{x}$ из $[\![\mathtt{A}]\!]$ получаем, что последовательность $\mathtt{f_n\,x}$, элементы которой принадлежат $[\![\mathtt{B}]\!]$, также возрастает, а значит, имеет предел. Обозначим через $\mathtt{F}$ функцию, которая ставит в соответствие элементу $\mathtt{x}$ предел последовательности $\mathtt{lim_n\,(f_n\,x)}$. Можно показать, хотя здесь мы этого делать не будем, что функция $\mathtt{F}$ принадлежит $[\![\mathtt{A} \to \mathtt{B}]\!]$, то есть является непрерывной (для этого необходима лемма о перестановке пределов). Функция $\mathtt{F}$ по построению оказывается больше всех функций $\mathtt{f_n}$, причём она наименьшая из всех таких функций. Следовательно, она является пределом последовательности $\mathtt{f_n}$. Любая возрастающая последовательность имеет предел, следовательно отношение порядка на $[\![\mathtt{A} \to \mathtt{B}]\!]$ является слабо полным.

Каждое множество $[\![\mathtt{A}]\!]$ имеет наименьший элемент, обозначаемый $\bot_{\mathtt{A}}$. Наименьший элемент $\mathtt{nat}$ это просто $\bot$, а наименьший элемент $[\![\mathtt{A} \to \mathtt{B}]\!]$ это постоянная функция, возвращающая значение $\bot_{\mathtt{B}}$ на любом аргументе.

5.3.4. Семантика неподвижной точки

Теперь можно вернуться к денотационной семантике PCF и добавить к определению отсутствующий **fix**:

- $[\![\mathtt{x}]\!]_{\mathtt{e}} = \mathtt{a}$, если $\mathtt{e}$ содержит определение $\mathtt{x = a}$;
- $[\![\mathbf{fun}\ \mathtt{x{:}A} \to \mathtt{t}]\!]_{\mathtt{e}} = \mathbf{fun}\ \mathtt{a{:}}[\![\mathtt{A}]\!] \to [\![\mathtt{t}]\!]_{\mathtt{e,x=a}}$;
- $[\![\mathtt{t\ u}]\!]_{\mathtt{e}} = [\![\mathtt{t}]\!]_{\mathtt{e}}[\![\mathtt{u}]\!]_{\mathtt{e}}$;
- $[\![\mathtt{n}]\!]_{\mathtt{e}} = \mathtt{n}$;
- $[\![\mathtt{t} \otimes \mathtt{u}]\!]_{\mathtt{e}} = [\![\mathtt{t}]\!]_{\mathtt{e}} \otimes [\![\mathtt{u}]\!]_{\mathtt{e}}$, если $[\![\mathtt{t}]\!]_{\mathtt{e}}$ и $[\![\mathtt{u}]\!]_{\mathtt{e}}$ натуральные числа, $\bot$ в противном случае;
- $[\![\mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v}]\!]_{\mathtt{e}} = [\![\mathtt{u}]\!]_{\mathtt{e}}$, если $[\![\mathtt{t}]\!]_{\mathtt{e}} = \mathtt{0}$, $[\![\mathtt{v}]\!]_{\mathtt{e}}$, если $[\![\mathtt{t}]\!]_{\mathtt{e}}$ — натуральное число, отличное от нуля, и $\bot_{\mathtt{A}}$, где $\mathtt{A}$ — тип этого терма, если $[\![\mathtt{t}]\!]_{\mathtt{e}} = \bot_{\mathtt{nat}}$;
- $[\![\mathbf{fix}\ \mathtt{x{:}A\ t}]\!]_{\mathtt{e}} = \mathtt{FIX}\ (\mathbf{fun}\ \mathtt{a{:}}[\![\mathtt{A}]\!]_{\mathtt{e}} \to [\![\mathtt{t}]\!]_{\mathtt{e,x=a}})$, где $\mathtt{FIX(f)}$ это наименьшая неподвижная точка непрерывной функции $\mathtt{f}$;
- $[\![\mathbf{let}\ \mathtt{x{:}A = t}\ \mathbf{in}\ \mathtt{u}]\!]_{\mathtt{e}} = [\![\mathtt{u}]\!]_{\mathtt{e,x=}[\![\mathtt{t}]\!]_{\mathtt{e}}}$.

Для доказательства корректности этого определения необходимо показать, что если $\mathtt{t}$ это терм типа $\mathtt{A}$, то $[\![\mathtt{t}]\!]$ принадлежит $[\![\mathtt{A}]\!]$, то есть, что функция $[\![\ ]\!]$ является непрерывной. Это действительно так, но доказывать этот факт мы здесь не будем.

Упражнение 5.7. Какова семантика терма **fun** x:nat → 0? Как насчёт термов **fix** x:nat x и (**fun** x:nat → 0) (**fix** x:nat x)?

Упражнение 5.8. Каково значение $[\![$**ifz** t **then** u **else** v$]\!]_e$, если $[\![t]\!]_e = 0$, $[\![u]\!]_e = 0$, $[\![v]\!]_e = \bot_{nat}$?

Теперь можно сформулировать теорему об эквивалентности двух семантик. Пусть t это замкнутый терм типа nat, n — натуральное число: $t \hookrightarrow n$ с вызовом по имени тогда и только тогда, когда $[\![t]\!] = n$. Прямое утверждение доказать нетрудно, а вот обратное далеко не тривиально.

Упражнение 5.9. Пользуясь теоремой об эквивалентности, покажите, что если t это замкнутый терм типа nat, такой что $[\![t]\!] = \bot$, то не существует такого натурального числа n, что $t \hookrightarrow n$.

Упражнение 5.10. Пусть G это денотационная семантика терма

fun f:(nat → nat) → **fun** n:nat → **ifz** n **then** 1 **else** n*(f (n - 1)).

Денотационная семантика терма

fix f:(nat → nat) **fun** n:nat → **ifz** n **then** 1 **else** n*(f (n - 1))

это наименьшая неподвижная точка G. По первой теореме о неподвижной точке существует предел последовательности $G^n(\bot_{nat\to nat})$. Что за функции обозначаются как $\bot_{nat\to nat}$ и $G^n(\bot_{nat\to nat})$? Установите предел этой последовательности.

Покажите, что для любого натурального числа p существует такое натуральное число m, что $G^m(\bot_{nat\to nat})(p) = \lim_n G^n(\bot_{nat\to nat})(p)$.

Упражнение 5.11. Рассмотрим следующие элементы множества функций $[\![$nat → nat$]\!]$: функцию u, отображающую $\bot$ на $\bot$, а все остальные элементы на 0; функцию v_i, отображающую $\bot$ на $\bot$, i на 1, а все остальные элементы на 0; и функцию w_i,отображающую $\bot$ на $\bot$, 0, 1, . . . , i - 1 на 0, а все остальные элементы на $\bot$.

Пусть F это возрастающая функция из $[\![$nat → nat$]\!]$ в $[\![$nat$]\!]$, такая что Fu = 0 и $Fv_i = 1$ для всех i. Покажите, что для всех i имеет место равенство $F\,w_i = \bot$. Покажите также, что функция F не является непрерывной.

Покажите, что в PCF невозможно написать такую функцию, которая принимает на вход функцию g типа nat → nat и возвращает 0 для всех n, таких что g n = 0, и 1 в противном случае.

Упражнение 5.12 (подход к непрерывности, основанный на информации)**.** Может показаться удивительным, что для определения семантики PCF, работающего не с вещественными, а только натуральными числами, понадобилось понятие непрерывности. На самом деле множество функций из $\mathbb{N}$ в $\mathbb{N}$ или множество последовательностей натуральных чисел очень близки к множеству вещественных чисел.

Интуитивно можно считать, что вещественная функция `f` является непрерывной, если для вычисления первых `n` знаков `f x` после запятой достаточно знать некоторое конечное число знаков `x`. К сожалению, технически это неверно, если `x` или `f x` имеют дробную часть. Будем говорить, что десятичная дробь приближает вещественное число до `n`-го знака после запятой, если расстояние между ними меньше, чем 10^{-n}. Например, число π имеет два приближения до двух знаков после запятой: `3,14` и `3,15`. Теперь можно сказать, что функция `f` является непрерывной, если для вычисления приближения `f x` до `n` знаков после запятой достаточно иметь некоторое десятичное приближение `x`.

Цель этого упражнения в том, чтобы показать, что функция `f`, отображающая последовательности натуральных чисел на последовательности натуральных чисел, является непрерывной, если для вычисления `n` первых элементов `f x` достаточно иметь некоторый начальный отрезок `x`. Если договориться называть конечный начальный отрезок последовательности конечным приближением, то это определение можно переформулировать следующим образом: для вычисления приближения `f x` с точностью до `n` элементов достаточно иметь некоторое приближение `x`.

Пусть `u` это последовательность натуральных чисел, а `U` это элемент множества $[\![\mathtt{nat} \to \mathtt{nat}]\!]$, отображающий $\bot$ на $\bot$, а $\mathtt{u_i}$ на `i`.

Пусть `V` это последовательность элементов из $[\![\mathtt{nat} \to \mathtt{nat}]\!]$:

$$[\bot\mapsto\bot, 0\mapsto\bot, 1\mapsto\bot, 2\mapsto\bot, 3\mapsto\bot, \ldots],$$

$$[\bot\mapsto\bot, 0\mapsto \mathtt{u_0}, 1\mapsto\bot, 2\mapsto\bot, 3\mapsto\bot, \ldots],$$

$$[\bot\mapsto\bot, 0\mapsto \mathtt{u_0}, 1\mapsto \mathtt{u_1}, 2\mapsto\bot, 3\mapsto\bot, \ldots],$$

$$[\bot\mapsto\bot, 0\mapsto \mathtt{u_0}, 1\mapsto \mathtt{u_1}, 2\mapsto \mathtt{u_2}, 3\mapsto\bot, \ldots],$$

$$\ldots$$

Покажите, что последовательность `V` сходится к `U`. Пусть `F` это непрерывная функция на $[\![\mathtt{nat} \to \mathtt{nat}]\!]$. Покажите, что последовательность $\mathtt{F\,V_i}$ сходится к `F U`. Покажите, что последовательность $\mathtt{F\,V_i\,p}$ сходится к `F U p`. Покажите, что существует такое натуральное число `k`, что $\mathtt{F\,V_k\,p = F\,U\,p}$. Покажите, что для вычисления `F U p` достаточно иметь первые `k` элемен-

тов U. Покажите, что для вычисления первых n элементов F U достаточно знать конечное число элементов U.

Рассмотрим функцию, отображающую последовательность u на число 0, если u состоит только из 0, и на 1 в противном случае. Является ли эта функция непрерывной? Можно ли её записать средствами PCF?

Наконец, заметьте, что в обоих этих примерах приближения — десятичные дроби и конечные последовательности — содержат конечный объём информации, тогда как объекты, которые они приближают — вещественные числа или бесконечные последовательности — бесконечный.

Упражнение 5.13 (System T Гёделя)**.** Во избежание незавершающихся вычислений можно заменить **fix** на конструкцию **rec**, позволяющую определять функции посредством индукции. Все программы на этом языке завершаются, однако он перестаёт быть полным по Тьюрингу. Тем не менее, довольно сложно найти программу, которую нельзя было бы представить на этом языке, для её построения необходимо быть экспертом в математической логике.

Функция f, определяемая равенствами f 0 = a и f (n+1) = g n (f n), записывается как **rec** a g. Правило операционной семантики с малым шагом для этой конструкции таково:

$$\textbf{rec}\ a\ g\ 0 \longrightarrow a$$

$$\textbf{rec}\ a\ g\ n \longrightarrow g\ (n-1)\ (\textbf{rec}\ a\ g\ (n-1)),$$

если n это отличное от 0 натуральное число.

Напишите на этом языке программу для вычисления факториала. Напишите правила типизации для **rec**. Определите для этого языка денотационную семантику.

Глава 6. Вывод типов

Во многих языках программирования, таких как Java или C, программисты обязаны указывать тип каждой из используемых в программе переменных. Например, они вынуждены всякий раз писать нечто подобное **fun** `x:nat` $\to$ `x + 1`. Однако, если известно, что операция `+` может работать только с числами, то нетрудно показать, что в терме **fun** `x` $\to$ `x + 1` переменная `x` должна принадлежать типу `nat`. Вместо того, чтобы просить программиста всюду указывать типы, можно было бы поручить компьютеру работу по их выводу. Именно в этом состоит цель алгоритма *вывода типов*.

6.1. Вывод мономорфных типов

6.1.1. Присвоение типов нетипизированным термам

Мы будет пользоваться исходным синтаксисом PCF, в котором переменным не приписываются типы. Вместо **fun** `x:nat` $\to$ `x + 1` будем снова, как во второй главе, писать **fun** `x` $\to$ `x + 1`.

Языки термов и типов теперь будут определяться независимо. Язык термов остаётся тем же, что был определён в гл. 2, а язык типов состоит из следующего:

— константа `nat`;

— символ $\to$ с двумя аргументами, не связывающий никакие переменные.

`A = X`

`| nat`

`| A` $\to$ `A`

Как и прежде, отношение `e` $\vdash$ `t : A` (читается «терм `t` имеет тип `A` в окружении `e`») может быть определено по индукции:

$$\frac{}{\texttt{e} \vdash \texttt{x:A}} \text{ если } \texttt{e} \text{ содержит } \texttt{x:A},$$

$$\frac{\texttt{e} \vdash \texttt{u:A} \qquad \texttt{e} \vdash \texttt{t:A} \to \texttt{B}}{\texttt{e} \vdash \texttt{t u:B}},$$

$$\frac{\texttt{(e, x:A)} \vdash \texttt{t:B}}{\texttt{e} \vdash \textbf{fun}\ \texttt{x} \to \texttt{t:A} \to \texttt{B}},$$

$$\frac{}{\texttt{e} \vdash \texttt{n:nat}},$$

$$\frac{\texttt{e} \vdash \texttt{u:nat} \qquad \texttt{e} \vdash \texttt{t:nat}}{\texttt{e} \vdash \texttt{t} \otimes \texttt{u:nat}},$$

$$\frac{\texttt{e} \vdash \texttt{t:nat} \qquad \texttt{e} \vdash \texttt{u:A} \qquad \texttt{e} \vdash \texttt{v:A}}{\texttt{e} \vdash \textbf{ifz}\ \texttt{t}\ \textbf{then}\ \texttt{u}\ \textbf{else}\ \texttt{v:A}},$$

$$\frac{\texttt{(e, x:A)} \vdash \texttt{t:A}}{\texttt{e} \vdash \textbf{fix}\ \texttt{x}\ \texttt{t:A}},$$

$$\frac{\texttt{e} \vdash \texttt{t:A} \qquad \texttt{(e, x:A)} \vdash \texttt{u:B}}{\texttt{e} \vdash \textbf{let}\ \texttt{x = t}\ \textbf{in}\ \texttt{u:B}}.$$

В этой системе типов некоторые термы, например **fun** x → x, могут иметь более одного типа. К примеру, мы можем вывести оба утверждения: ⊢ **fun** x → x : nat → nat и ⊢ **fun** x → x : (nat → nat) → (nat → nat). Замкнутый терм может иметь тип со свободными переменными, например, терм **fun** x → x имеет в пустом окружении тип X → X.

Можно доказать, что если замкнутый терм t имеет тип A, содержащий при пустом окружении переменные, то t имеет также тип θA для любой подстановки θ. Например, при подстановке вместо переменной X типа nat → nat в X → X получим тип (nat → nat) → (nat → nat), а это один из допустимых типов для терма **fun** x → x.

6.1.2. Алгоритм Хиндли

Теперь можно описать алгоритм вывода типов. Начнём с описания двухэтапной версии алгоритма. Первый этап похож на алгоритм проверки типов: на нём организуется рекурсивный обход терма, проверяется, удовлетворяются ли типовые ограничения, и вычисляется тип терма. Есть, правда, два важных отличия: во-первых, при попытке приписывания типа терму вида **fun** x → t в окружении e, поскольку тип переменной x нам неизвестен, приходится создавать типовую переменную X, расширять окружение e объявлением x:X и типизировать терм t в этом расширенном окружении. Во-вторых, при типизации применения t u после вычисления типов A и B для u и t соответственно нельзя легко проверить, что B имеет вид A → C. Действительно, эти типы могут содержать переменные. По этой

причине в этом месте создаётся уравнение, связывающее типы: `B = A → X`. На втором этапе вывода типов алгоритм разрешает полученные уравнения.

Проиллюстрируем эти идеи на примере: чтобы определить тип терма **fun** `f → 2 + (f 1)`, нужно типизировать терм `2 + (f 1)` в окружении `f : X`. Для этого необходимы типы терма `2`, то есть `nat`, и терма `f 1`. Терм `1` имеет тип `nat`, а терм `f` — тип `X`. Создаём уравнение `X = nat → Y`, и типом `f 1` теперь оказывается `Y`. Поскольку термы `2` и `f 1` уже типизированы, создаём уравнения `nat = nat` и `Y = nat`, теперь тип терма `2 + (f 1)` это `nat`. Наконец, тип всего терма **fun** `f → 2 + (f 1)` это `X → nat`, и вот уравнения, которые необходимо решить:

`X = nat → Y`

`nat = nat`

`Y = nat`

Эта система уравнений имеет единственное решение `X = nat → nat`, `Y = nat`, следовательно единственный тип, который можно присвоить терму **fun** `f → 2 + (f 1)`, это `(nat → nat) → nat`.

Первый этап алгоритма можно описать с помощью набора правил в стиле операционной семантики с большим шагом (то же самое мы делали для алгоритма проверки типов), но в этом случае результатом интерпретации терма будет не значение или тип, а пара: тип и множество уравнений на типах. Для обозначения отношения между окружением `e`, термом `t`, типом `A` и множеством уравнений `E` будем использовать запись $\mathtt{e} \vdash \mathtt{t} \rightsquigarrow \mathtt{A},\ \mathtt{E}$.

$$\frac{}{\mathtt{e} \vdash \mathtt{x} \rightsquigarrow \mathtt{A},\ \varnothing} \text{ если } \mathtt{e} \text{ содержит } \mathtt{x{:}A},$$

$$\frac{\mathtt{e} \vdash \mathtt{u} \rightsquigarrow \mathtt{A},\ \mathtt{E} \qquad \mathtt{e} \vdash \mathtt{t} \rightsquigarrow \mathtt{B},\ \mathtt{F}}{\mathtt{e} \vdash \mathtt{t\ u} \rightsquigarrow \mathtt{X},\ \mathtt{E} \cup \mathtt{F} \cup \{\mathtt{B} = \mathtt{A} \to \mathtt{X}\}},$$

$$\frac{(\mathtt{e}, \mathtt{x{:}X}) \vdash \mathtt{t} \rightsquigarrow \mathtt{A},\ \mathtt{E}}{\mathtt{e} \vdash \mathbf{fun}\ \mathtt{x} \to \mathtt{t} \rightsquigarrow \mathtt{X} \to \mathtt{A},\ \mathtt{E}},$$

$$\frac{}{\mathtt{e} \vdash \mathtt{n} \rightsquigarrow \mathtt{nat},\ \varnothing},$$

$$\frac{\mathtt{e} \vdash \mathtt{u} \rightsquigarrow \mathtt{A},\ \mathtt{E} \qquad \mathtt{e} \vdash \mathtt{t} \rightsquigarrow \mathtt{B},\ \mathtt{F}}{\mathtt{e} \vdash \mathtt{t} \otimes \mathtt{u} \rightsquigarrow \mathtt{nat},\ \mathtt{E} \cup \mathtt{F} \cup \{\mathtt{A} = \mathtt{nat}, \mathtt{B} = \mathtt{nat}\}},$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \rightsquigarrow \mathtt{A},\ \mathtt{E} \qquad \mathtt{e} \vdash \mathtt{u} \rightsquigarrow \mathtt{B},\ \mathtt{F} \qquad \mathtt{e} \vdash \mathtt{v} \rightsquigarrow \mathtt{C},\ \mathtt{G}}{\mathtt{e} \vdash \mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v} \rightsquigarrow \mathtt{B},\ \mathtt{E} \cup \mathtt{F} \cup \mathtt{G} \cup \{\mathtt{A} = \mathtt{nat}, \mathtt{B} = \mathtt{C}\}},$$

$$\frac{(\mathtt{e}, \mathtt{x{:}X}) \vdash \mathtt{t} \rightsquigarrow \mathtt{A},\ \mathtt{E}}{\mathtt{e} \vdash \mathbf{fix}\ \mathtt{x}\ \mathtt{t} \rightsquigarrow \mathtt{A},\ \mathtt{E} \cup \{\mathtt{X} = \mathtt{A}\}},$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \rightsquigarrow \mathtt{A},\ \mathtt{E} \qquad (\mathtt{e}, \mathtt{x{:}A}) \vdash \mathtt{u} \rightsquigarrow \mathtt{B},\ \mathtt{F}}{\mathtt{e} \vdash \mathbf{let}\ \mathtt{x} = \mathtt{t}\ \mathbf{in}\ \mathtt{u} \rightsquigarrow \mathtt{B},\ \mathtt{E} \cup \mathtt{F}}.$$

В правиле для применения переменная X это произвольная переменная, не встречающаяся в e, A, B, E и F. В правилах для **fun** и **fix** это произвольная переменная, не встречающаяся в e.

Пусть t это замкнутый терм, A и E — тип и система уравнений, полученных алгоритмом, то есть $\vdash \mathtt{t} \rightsquigarrow \mathtt{A},\ \mathtt{E}$. Подстановка $\sigma = \mathtt{B}_1/\mathtt{X}_1, \ldots, \mathtt{B}_n/\mathtt{X}_n$ называется *решением* E, если для каждого уравнения C = D из E типы σC и σD совпадают. Можно показать, что если подстановка σ является решением для множества E, то типом терма t в пустом окружении является тип σA. И вообще, если $\mathtt{e} \vdash \mathtt{t} \rightsquigarrow \mathtt{A},\ \mathtt{E}$, то для любого решения σ множества уравнений E тип σA является типом терма t в окружении σe. Верно и обратное: если $\mathtt{A}'$ это тип терма t в пустом окружении, то существует такая подстановка σ, что $\mathtt{A}' = \sigma\mathtt{A}$ и σ является решением множества уравнений E.

Второй этап алгоритма имеет дело с уравнениями на типах. Язык типов не содержит связывающих конструкций, это язык, порождённый константой nat и двухаргументным символом $\to$. Для разрешения множества уравнений мы воспользуемся алгоритмом унификации Робинсона, который способен решать уравнения в любом языке без связываний. В некоторых отношениях этот алгоритм похож на алгоритм Гаусса решения систем линейных алгебраических уравнений. Он состоит из последовательности следующих преобразований:

— уравнение вида $\mathtt{A} \to \mathtt{B} = \mathtt{C} \to \mathtt{D}$ заменяется уравнениями A = C и B = D;

— уравнение вида nat = nat удаляется из системы уравнений;

— если в системе уравнений имеется уравнение вида $\mathtt{nat} = \mathtt{A} \to \mathtt{B}$ или $\mathtt{A} \to \mathtt{B} = \mathtt{nat}$, то алгоритм завершается с ошибкой;

— уравнение вида X = X удаляется из системы уравнений;

— если в системе уравнений имеется уравнение вида X = A или A = X, где X входит в A, причём A отлично от X, то алгоритм завершается с ошибкой;

— если в системе уравнений имеется уравнение вида X = A или A = X, где X не входит в A, но при этом входит в другие уравнения системы, то X заменяется на A во всех уравнениях системы.

Этот алгоритм завершается, хотя доказательство соответствующего факта нетривиально. Если он завершается с ошибкой, то система не имеет решения. Если он завершается без ошибки, то окончательная система имеет вид $\mathtt{X_1 = A_1, \ldots, X_n = A_n}$, где $\mathtt{X_i}$ это различные переменные, не входящие в $\mathtt{A_i}$. В этом случае решением исходной системы является подстановка $\sigma = \mathtt{A_1/X_1, \ldots, A_n/X_n}$. Можно доказать, что эта подстановка является *главным* (*principal*) решением системы, или, другими словами, что для любого решения θ исходной системы имеется некоторая подстановка η, такая что $\theta = \eta \circ \sigma$. Мы пишем $\sigma = \mathtt{mgu(E)}$ — *наиболее общий унификатор* (*most general unifier*) — главное решение.

Пусть `t` замкнутый терм, `A` и `E` таковы, что $\vdash \mathtt{t} \rightsquigarrow \mathtt{A, E}$. Пусть σ — главное решение `E`. Тогда терм `t` в пустом окружении имеет тип $\sigma\mathtt{A}$. Более того, $\sigma\mathtt{A}$ является *главным* типом `t`, то есть для любого другого типа `B` терма `t` существует такая подстановка η, что $\mathtt{B} = \eta\sigma\mathtt{A}$.

6.1.3. Алгоритм Хиндли с немедленным разрешением

Имеется вариант алгоритма Хиндли, в котором вместо того, чтобы ждать завершения первого этапа и только потом приступать к разрешению уравнений, уравнения решаются по мере добавления. В этом случае алгоритм возвращает не тип и набор уравнений, как прежде, а тип вместе с подстановкой ρ, являющейся главным решением системы уравнений. Можно даже применять подстановку ρ к типу `A` по мере её построения.

Алгоритм имеет следующее свойство: если $\mathtt{e} \vdash \mathtt{t} \rightsquigarrow \mathtt{A,}\ \rho$, то `A` это главный тип `t` в окружении $\rho\mathtt{e}$. Приведём определение этого алгоритма:

$$\frac{}{\mathtt{e} \vdash \mathtt{x} \rightsquigarrow \mathtt{A,}\ \varnothing} \text{ если } \mathtt{e} \text{ содержит } \mathtt{x{:}A},$$

$$\frac{\mathtt{e} \vdash \mathtt{u} \rightsquigarrow \mathtt{A,}\ \rho \qquad \rho\mathtt{e} \vdash \mathtt{t} \rightsquigarrow \mathtt{B,}\ \rho'}{\mathtt{e} \vdash \mathtt{t\ u} \rightsquigarrow \sigma\mathtt{X,}\ \sigma \circ \rho' \circ \rho} \text{ если } \sigma = \mathtt{mgu(B} = \rho'\mathtt{A} \to \mathtt{X)},$$

$$\frac{\mathtt{(e, x{:}X)} \vdash \mathtt{t} \rightsquigarrow \mathtt{A,}\ \rho}{\mathtt{e} \vdash \textbf{fun}\ \mathtt{x} \to \mathtt{t} \rightsquigarrow \mathtt{(}\rho\mathtt{X} \to \mathtt{A),}\ \rho},$$

$$\frac{}{\mathtt{e} \vdash \mathtt{n} \rightsquigarrow \mathtt{nat,}\ \varnothing},$$

$$\frac{\mathtt{e} \vdash \mathtt{u} \rightsquigarrow \mathtt{A,}\ \rho \qquad \sigma\rho\mathtt{e} \vdash \mathtt{t} \rightsquigarrow \mathtt{B,}\ \rho'}{\mathtt{e} \vdash \mathtt{t} \otimes \mathtt{u} \rightsquigarrow \mathtt{nat,}\ \sigma' \circ \rho' \circ \sigma \circ \rho} \text{ если } \begin{bmatrix} \sigma = \mathtt{mgu(A} = \mathtt{nat)}, \\ \sigma' = \mathtt{mgu(B} = \mathtt{nat)}, \end{bmatrix}$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \rightsquigarrow \mathtt{A},\ \rho \qquad \sigma\rho\mathtt{e} \vdash \mathtt{u} \rightsquigarrow \mathtt{B},\ \rho' \qquad \rho'\sigma\rho\mathtt{e} \vdash \mathtt{v} \rightsquigarrow \mathtt{C},\ \rho''}{\mathtt{e} \vdash \textbf{ifz t then u else v} \rightsquigarrow \sigma'\mathtt{C},\ \sigma' \circ \rho'' \circ \rho' \circ \sigma \circ \rho}$$

если $\sigma = \mathtt{mgu(A = nat)}$ и $\sigma' = \mathtt{mgu}(\rho''\mathtt{B} = \mathtt{C})$,

$$\frac{\mathtt{(e, x:X)} \vdash \mathtt{t} \rightsquigarrow \mathtt{A},\ \rho}{\mathtt{e} \vdash \textbf{fix x t} \rightsquigarrow \sigma\mathtt{A},\ \sigma \circ \rho} \text{ если } \sigma = \mathtt{mgu(A} = \rho\mathtt{X)},$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \rightsquigarrow \mathtt{A},\ \rho \qquad (\rho\mathtt{e}, \mathtt{x:A}) \vdash \mathtt{u} \rightsquigarrow \mathtt{B},\ \rho'}{\mathtt{e} \vdash \textbf{let x = t in u} \rightsquigarrow \mathtt{B},\ \rho' \circ \rho}.$$

Снова в правиле для применения X это произвольная переменная, не входящая в e, A, B, ρ и ρ', а в правилах для **fun** и **fix** это переменная, не входящая в e.

Упражнение 6.1.

(1) Определите главный тип терма **fun** x → **fun** y → (x (y + 1)) + 2. Опишите все его типы.

(2) Определите главный тип терма **fun** x → x. Опишите все его типы.

Упражнение 6.2 (единственность главного типа)**.** Подстановка σ называется *переименованием*, если она является инъективным отображением, ставящим в соответствие переменной переменную. Переименованием, к примеру, будет подстановка y/x, z/y.

(1) Пусть A это тип, а σ и σ' — две подстановки. Покажите, что если $\sigma'\sigma\mathtt{A} = \mathtt{A}$, то $\sigma|_{\mathtt{FV(A)}}$ является переименованием.

(2) Докажите, что если A и A′ это два главных типа терма t, то существует переименование θ, определённое на FV(A), такое что $\mathtt{A}' = \theta\mathtt{A}$.

Упражнение 6.3. В случае произвольного языка без связываний первые три правила алгоритма унификации Робинсона можно заменить следующими двумя:

— уравнение вида $\mathtt{f}(\mathtt{u}_1, \ldots, \mathtt{u}_n) = \mathtt{f}(\mathtt{v}_1, \ldots, \mathtt{v}_n)$ заменяется на уравнения $\mathtt{u}_1 = \mathtt{v}_1, \ldots, \mathtt{u}_n = \mathtt{v}_n$;

— если в системе имеется уравнение вида $\mathtt{f}(\mathtt{u}_1, \ldots, \mathtt{u}_n) = \mathtt{g}(\mathtt{v}_1, \ldots, \mathtt{v}_n)$, где f и g — различные символы, то алгоритм завершается с ошибкой.

(1) Имеет ли уравнение (2 + (3 + X)) = (X + (Y + 2)) решение в языке, состоящем из двухаргументного символа + и целочисленных констант?

(2) Чем отличаются уравнения в этом языке от изучавшихся в школе уравнений в целых числах?

(3) Пользуясь операционной семантикой PCF с малым шагом, определите школьное понятие уравнения. Имеет ли в этом случае решение уравнение `X + 2 = 4`?

6.2. Полиморфизм

Мы уже видели, что главный тип терма `id = fun x → x` это `X → X`. Это означает, что терм `id` имеет тип `A → A` для любого типа `A`. Мы можем приписать такому терму новый тип `∀X (X → X)` и добавить правило, что если терм `t` имеет тип `∀X A`, то он имеет тип `(B/X)A` для любого типа `B`. Язык типов, включающий квантор всеобщности, называется *полиморфным*.

В системе, представленной в предыдущем разделе, было невозможно приписать тип терму **let** `id = fun x → x` **in** `id id`. В самом деле, правило типизации для **let** требует, чтобы мы предварительно типизировали оба терма **fun** `x → x` и `idid`, однако последний терм типизировать невозможно, потому что мы не можем присвоить один и тот же тип обоим вхождениям `id`. Именно поэтому нельзя приписать никакой тип терму целиком. Эту ситуацию можно рассматривать как дефект системы типизации, ведь терм `(fun x → x)(fun x → x)`, полученный заменой `id` на его определение, типизируем. Действительно, для этого достаточно присвоить тип `nat → nat` первому вхождению связанной переменной `x` и тип `nat` второму.

Если же мы присвоим тип `∀X (X → X)` символу `id` в терме **let** `id = fun x → x` **in** `idid`, то для каждого вхождения `id` в терм `idid` можно будет воспользоваться разными типами, и терм станет типизируемым.

Типизация терма **let** `id = fun x → x` **in** `id id` может показаться не таким уж важным вопросом, чтобы платить за него столь высокую цену, как введение в язык кванторов, получая взамен незначительный прирост выразительности. Однако это впечатление обманчиво. На самом деле в расширении PCF списками (см. упр. 3.14) именно это добавка позволяет разработать единственный алгоритм сортировки и применять его к любым спискам независимо от типов их аргументов: **let** `sort = t` **in** `u`. Полиморфизм способствует повторному использованию кода, а значит, позволяет писать более лаконичные программы.

Теперь мы будем приписывать тип с квантором переменным, связываемым в **let**, используя при этом обычный тип для переменных, связанных **fun** и **fix**.

6.2.1. PCF с полиморфными типами

Необходимо различать типы без кванторов — мы будем по-прежнему называть их *типами* — и типы с кванторами, которые назовём *схемами типа*. Схема имеет вид $\forall \mathtt{X_1} \ldots \forall \mathtt{X_n}\ \mathtt{A}$, где `A` это тип.

Затем мы определим типовый язык с двумя сортами: сорт для типов и сорт для схем. Поскольку в многосортном языке множества термов каждого сорта не пересекаются, множество типов не может быть подмножеством множества схем, поэтому нам потребуется символ `[ ]`, с помощью которого мы будем включать тип в сорт схем. Таким образом, если `A` это тип, то `[A]` будет схемой, состоящей из типа `A` без переменных, связанных кванторами.

Язык типов и схем определяется следующим:

— константа типа `nat`;

— символ типов $\to$ с двумя аргументами, не связывающий никакие переменные;

— символ схем `[ ]` с одним аргументом, не связывающий никакие переменные;

— символ схем $\forall$ с одним аргументом, связывающий переменную в своём аргументе.

`A` = `X`

| `nat`

| `A` $\to$ `A`

`S` = `Y`

| `[A]`

| $\forall$`X S`

Этот язык включает переменные обоих сортов, в частности, переменные схем. Однако эти переменные использоваться не будут.

Окружение теперь это список, связывающий схемы с переменными. Определим посредством индукции отношение «терм `t` имеет схему `S` в окружении `e`»:

$$\frac{}{\mathtt{e} \vdash \mathtt{x{:}S}}$$ если `e` содержит `x:S`,

$$\frac{\mathtt{e} \vdash \mathtt{u}:[\mathtt{A}] \qquad \mathtt{e} \vdash \mathtt{t}:[\mathtt{A} \to \mathtt{B}]}{\mathtt{e} \vdash \mathtt{t}\ \mathtt{u}:[\mathtt{B}]},$$

$$\frac{(\mathtt{e}, \mathtt{x}:[\mathtt{A}]) \vdash \mathtt{t}:[\mathtt{B}]}{\mathtt{e} \vdash \mathbf{fun}\ \mathtt{x} \to \mathtt{t}:[\mathtt{A} \to \mathtt{B}]},$$

$$\frac{}{\mathtt{e} \vdash \mathtt{n}:[\mathtt{nat}]},$$

$$\frac{\mathtt{e} \vdash \mathtt{u}:[\mathtt{nat}] \qquad \mathtt{e} \vdash \mathtt{t}:[\mathtt{nat}]}{\mathtt{e} \vdash \mathtt{t} \otimes \mathtt{u}:[\mathtt{nat}]},$$

$$\frac{\mathtt{e} \vdash \mathtt{t}:[\mathtt{nat}] \qquad \mathtt{e} \vdash \mathtt{u}:[\mathtt{A}] \qquad \mathtt{e} \vdash \mathtt{v}:[\mathtt{A}]}{\mathtt{e} \vdash \mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v}:[\mathtt{A}]},$$

$$\frac{(\mathtt{e}, \mathtt{x}:[\mathtt{A}]) \vdash \mathtt{t}:[\mathtt{A}]}{\mathtt{e} \vdash \mathbf{fix}\ \mathtt{x}\ \mathtt{t}:[\mathtt{A}]},$$

$$\frac{\mathtt{e} \vdash \mathtt{t}:\mathtt{S} \qquad (\mathtt{e}, \mathtt{x}:\mathtt{S}) \vdash \mathtt{u}:[\mathtt{B}]}{\mathtt{e} \vdash \mathbf{let}\ \mathtt{x} = \mathtt{t}\ \mathbf{in}\ \mathtt{u}:[\mathtt{B}]},$$

$$\frac{\mathtt{e} \vdash \mathtt{t}:\mathtt{S}}{\mathtt{e} \vdash \mathtt{t}:\forall \mathtt{X}\ \mathtt{S}} \text{ если } \mathtt{X} \text{ не входит свободно в } \mathtt{e},$$

$$\frac{\mathtt{e} \vdash \mathtt{t}:\forall \mathtt{X}\ \mathtt{S}}{\mathtt{e} \vdash \mathtt{t}:(\mathtt{A}/\mathtt{X})\mathtt{S}}.$$

Индуктивное определение присваивает каждому терму, в частности, переменным, схему. Именно поэтому переменные связываются в окружении со схемами. Однако, когда мы типизируем терм вида **fun** `x` $\to$ `t` или **fix** `x t`, мы определяем тип `t` в расширенном окружении, где переменная `x` связана со схемой `[A]` без кванторов. Схему можно ассоциировать с термом `t` только в процессе типизации терма вида **let** `x = t` **in** `u`, когда эта схема связывается с переменной `x`.

Для введения в схему, связанную с термом `t`, кванторов используется предпоследнее правило, позволяющее навесить квантор на переменную в схеме `S` при условии, что эта переменная не входит свободно в `e`. Таким образом, в пустом окружении после присвоения схемы `[X` $\to$ `X]` терму **fun** `x` $\to$ `x` мы можем присвоить ему схему $\forall$`X [X` $\to$ `X]`. Заметьте, что в окружении `x : [X]` после присвоения схемы `[X]` переменной `x` присвоить схему $\forall$`X [X]` нельзя.

Наконец, заметьте, что если мы присвоили переменной или произвольному терму схему с кванторами, то квантор можно удалить, подставив,

пользуясь последним правилом, свободную переменную. Например, в окружении $\mathtt{x}{:}\forall \mathtt{X}\ [\mathtt{X} \to \mathtt{X}]$ переменной $\mathtt{x}$ можно присвоить схему $[\mathtt{nat} \to \mathtt{nat}]$.

6.2.2. Алгоритм Дамаса—Милнера

Теперь мы готовы определить алгоритм вывода типов. Будем решать уравнения на лету, как это делалось во втором варианте алгоритма Хиндли. Алгоритм будет применяться к терму $\mathtt{t}$ и окружению $\mathtt{e}$ и возвращать тип $\mathtt{A}$ вместе с подстановкой ρ, такой что терм $\mathtt{t}$ имеет схему $[\mathtt{A}]$ в окружении $\rho\mathtt{e}$. Единственное отличие от второго варианта алгоритма Хиндли в первых двух правилах:

$$\frac{}{\mathtt{e} \vdash \mathtt{x} \rightsquigarrow (\mathtt{Y_1}/\mathtt{X_1} \ldots \mathtt{Y_n}/\mathtt{X_n})\mathtt{A},\ \varnothing}$$

если $\mathtt{e}$ содержит $\forall \mathtt{X_1} \ldots \forall \mathtt{X_n}\ [\mathtt{A}]$ и $\mathtt{Y_1}, \ldots, \mathtt{Y_n}$ — новые переменные,

$$\frac{\mathtt{e} \vdash \mathtt{t} \rightsquigarrow \mathtt{A},\ \rho \qquad (\rho\mathtt{e}, \mathtt{x}{:}\mathtt{Gen}(\mathtt{A}, \rho\mathtt{e})) \vdash \mathtt{u} \rightsquigarrow \mathtt{B},\ \rho'}{\mathtt{e} \vdash \textbf{let}\ \mathtt{x} = \mathtt{t}\ \textbf{in}\ \mathtt{u} \rightsquigarrow \mathtt{B},\ \rho' \circ \rho},$$

где $\mathtt{Gen}(\mathtt{A}, \mathtt{e})$ это схема, полученная из $[\mathtt{A}]$ навешиванием кванторов на все типовые переменные, входящие свободно в $[\mathtt{A}]$, но не в $\mathtt{e}$.

Можно доказать, что если терм $\mathtt{t}$ замкнут, то тип $\mathtt{A}$, вычисленный этим алгоритмом, является главным типом $\mathtt{t}$, то есть, если $\vdash \mathtt{t}{:}[\mathtt{B}]$, то $\mathtt{B}$ это экземпляр $\mathtt{A}$.

Упражнение 6.4. Рассмотрим расширение PCF символом типа `list` с одним типовым аргументом. Мы пишем `nat list` для типа списка натуральных чисел, `(nat → nat) list` будет типом списков функций на натуральных числах, а `(nat list) list` — типом списков, элементами которых являются списки натуральных чисел.

Добавим в язык следующие конструкции:

— константа `nil` типа `(A list)` для любого типа `A`, представляющая пустой список;

— символ `cons a l` типа `(A list)` для любого типа `A`, где `a` имеет тип `A`, а `l` — тип `A list`, представляющий список, первый элемент которого `a`, а `l` это список остальных элементов;

— символ **ifnil** `t` **then** `u` **else** `v` типа `A` при условии, что `t` имеет тип `B list`, `u` и `v` — термы типа `A`, предназначенный для проверки, пуст ли список `t` или нет;

— символ `hd l` типа `A`, если `l` типа `A list`, возвращающий первый элемент списка `l`;

— символ `tl l` типа `A list`, если `l` типа `A list`, возвращающий список `l` без первого элемента.

(1) Напишите правила типизации и реализуйте проверку типов для этого расширения.

(2) Запрограммируйте на этом языке функцию `map`, отображающую функцию `f` и список $\mathtt{t}_1, \ldots, \mathtt{t}_n$ на список $\mathtt{f}\ \mathtt{t}_1, \ldots, \mathtt{f}\ \mathtt{t}_n$. Каков тип этой функции?

(3) Запрограммируйте алгоритм сортировки. Каков тип этого алгоритма?

В описанной здесь системе типов допускается использование кванторов для переменных, связанных в **let**. Можно было бы попытаться делать то же самое для переменных, связанных **fun**. Например, мы могли бы присвоить тип $\forall \mathtt{X}\ (\mathtt{X} \to \mathtt{X})$ переменной `x` в терме **fun** `x` $\to$ `x x`, что позволило бы его типизировать. Язык, полученный таким образом, называется системой F (System F), он был определён Жираром (Girard) и Рейнольдсом (Reynolds). Правда, как показал Уэллс (Wells), отношение типизации в System F неразрешимо, поэтому не стоит надеяться на получение в этой системе алгоритма вывода типов. Похожим образом, если позволить переменной, связанной **fix**, быть полиморфной, система снова становится неразрешимой, что доказал Кфури (Kfoury). Ограничение полиморфных возможностей системы конструкцией **let** можно рассматривать как хороший компромисс: он предлагает приемлемый уровень повторного использования и вывод типов.

Глава 7. Ссылки и присваивание

Рассмотрим два примера: число π и температуру в Париже. Сегодня число π имеет значение между `3,14` и `3,15`, а температура в Париже — между `16` и `17` градусами. Завтра число π будет иметь то же самое значение, а вот температура в Париже наверняка изменится. В математике *числа* представляют собой сущности, значения которых со временем не меняются: температура в Париже это не изменяющееся число, это функция, значение которой меняется со временем.

Однако формализация температуры как функции от времени имеет, вероятно, слишком общий характер. Она не позволяет учесть тот факт, что изменение во времени температуры в данном месте зависит, вообще говоря, не от температуры десять секунд до или после, а от температуры в данном месте. Получается, что системе доступна не полная температурная функция, а только лишь её текущее значение. Именно поэтому физические уравнения обычно оказываются не произвольными уравнениями над функциями, а дифференциальными уравнениями.

В информатике программам также необходимы сущности, которые меняются со временем. Например, в программе, управляющей продажей билетов на концерт, число мест со временем меняется: оно уменьшается всякий раз, когда кто-то покупает билет. С математической точки зрения это функция от времени. Однако, чтобы знать, можно ли продать билет, программе достаточно знать текущее значение функции, а вовсе не всю функцию: в некоторый конкретный момент времени `t` достаточно знать значение функции в момент `t`.

По этой причине при написании такой программы мы представляем количество мест не функцией, то есть не термом вида $\texttt{nat} \to \texttt{nat}$ — в предположении, что время дискретно, — что означало бы, что в каждый момент времени `t` мы знаем имеющееся число мест на любое другое время $\texttt{t}'$. Ясно, что это вообще невозможно, поскольку для этого требуется знать количество мест в некоторый момент $\texttt{t}'$ в будущем. Мы также не можем выразить это число термом типа `nat`, потому что значение терма типа `nat` в PCF, то есть число, не может меняться со временем. Поэтому для представления значений, меняющихся со временем, придётся ввести другой сорт термов: *ссылки* (*reference*), называемые также *переменными*, мы, правда, предпочитаем это второе название в таком контексте не использовать, поскольку

смысл термина «ссылка» слишком сильно отличается от смысла термина «переменная» в математике и в функциональных языках.

Если `x` это ссылка, то с ней можно делать две вещи: получать текущее значение `!x` и изменять значение `x := t`, то есть участвовать в построении функции, которая упоминалась выше, утверждая тот факт, что отныне и до следующего указания текущим значением будет значение терма `t`.

Довольно тонким является вопрос равенства «чисел, меняющихся во времени». Можно было бы сравнить такое число, как температура в Париже, с листом дерева: маленький, зелёный и эластичный весной, он становится большим, жёлтым и хрупким осенью. Очевидно, что он меняется, но мы знаем, что это тот же самый лист: никто бы не поверил, что маленький зелёный лист вдруг исчез, а из ничего возник большой жёлтый. Хотя преобразование и происходит, тот же лист остаётся на дереве с марта по октябрь. Это пример старого парадокса, что нечто, меняясь, остаётся тем же самым. Аналогично и суть парижской температуры остаётся прежней, хотя температура меняется со временем. С другой стороны, парижскую температуру нетрудно отличить от римской: это две разные сущности, однако температура в обоих городах порой оказывается одинаковой.

Одним из способов разобраться с этим парадоксом был бы такой: рассматривать температуру в Париже и температуру в Риме как функции — функция может принимать разные значения в разных точках, оставаясь при этом той же функцией, две же различные функции вполне могут в некоторой точке принимать одинаковые значения.

В программе, если `x` и `y` это две ссылки, и нам нужно их сравнить, следует отличать их равенство как ссылок, то есть совпадают ли они как сущности или, математически, являются ли они одной и той же функцией от времени, и равенство их значений, то есть совпадают ли числа `!x` и `!y` в некоторый конкретный момент времени. В частности, из равенства ссылок следует, что при изменении значения `x` одновременно меняется значение `y`, если же это разные ссылки с одинаковыми значениями, то ничего подобного не происходит.

7.1. Расширение PCF

Расширим язык PCF двумя новыми конструкциями `!` и `:=`.

Терм `x := 4` обозначает следующее действие: значение, связанное со ссылкой `x`, обновляется. Сравните это с уже встречавшимся нам термом `fact 3`, который тоже обозначает действие: вычисление факториала числа `3`. Вот отличие: результатом (эффектом) вычисления факториала `3` является значение, тогда как результатом (эффектом) действия `x := 4` яв-

ляется изменение «глобального состояния» Вселенной. До этого действия ссылка x имела, скажем, значение 0, а после него — значение 4. После добавления в PCF ссылок результатом интерпретации терма становится не просто значение, а значение вместе с новым состоянием Вселенной. Это изменение состояния называют *побочным эффектом* (*side effect*) интерпретации терма.

Формальная семантика ссылок в PCF определяет глобальное состояние как функцию из конечного множества R во множество значений PCF-термов. Элементы множества R называются *ссылками*. В *машинном*, родном для компьютера, языке множество ссылок фиксировано, это просто множество адресов памяти. В других языках множество R произвольно. В частности, при определении семантики языка мы не различаем множества R и $\mathtt{R}'$ одинаковой мощности (то есть с одним и тем же количеством элементов). Это соответствует тому, что программисты не знают точные адреса в памяти, по которым будут храниться данные.

В PCF, как и в большинстве других языков программирования, значения, связанные со ссылками, могут со временем изменяться. Более того, само множество R может меняться: во время выполнения программы вполне допустимо создание новых ссылок. Для этого в язык добавляется конструкция **ref**. Побочным эффектом интерпретации терма **ref** t является создание новой ссылки, начальным значением которой является значение терма t. Значением, вычисляемым интерпретацией этой конструкции, является сама ссылка.

Поскольку интерпретация терма **ref** t производит значение, являющееся ссылкой, ясно, что в этом расширении PCF значениями могут быть ссылки.

7.2. Семантика PCF со ссылками

В операционной семантике с большим шагом для этого расширения соответствующее отношение имеет вид $\mathtt{e,\ m} \vdash \mathtt{t} \hookrightarrow \mathtt{V,\ m}'$, где t это интерпретируемый терм, e — окружение, в котором он интерпретируется, m — глобальное состояние, в котором происходит интерпретация, V — значение, полученное в результате интерпретации, $\mathtt{m}'$ — новое глобальное состояние по итогам интерпретации:

$$\frac{}{\mathtt{e,\ m} \vdash \mathtt{x} \hookrightarrow \mathtt{V,\ m}} \text{ если e содержит } \mathtt{x = V},$$

$$\frac{\mathtt{e'},\ \mathtt{m} \vdash \mathbf{fix}\ \mathtt{y}\ \mathtt{t} \hookrightarrow \mathtt{V},\ \mathtt{m'}}{\mathtt{e},\ \mathtt{m} \vdash \mathtt{x} \hookrightarrow \mathtt{V},\ \mathtt{m'}}\ \text{если } \mathtt{e} \text{ содержит } \mathtt{x} = \langle \mathbf{fix}\ \mathtt{y}\ \mathtt{t}, \mathtt{e'} \rangle,$$

$$\frac{\mathtt{e},\ \mathtt{m} \vdash \mathtt{u} \hookrightarrow \mathtt{W},\ \mathtt{m'} \qquad \mathtt{e},\ \mathtt{m}' \vdash \mathtt{t} \hookrightarrow \langle \mathtt{x}, \mathtt{t'}, \mathtt{e'} \rangle,\ \mathtt{m''} \qquad (\mathtt{e'}, \mathtt{x} = \mathtt{W}),\ \mathtt{m''} \vdash \mathtt{t'} \hookrightarrow \mathtt{V},\ \mathtt{m'''}}{\mathtt{e},\ \mathtt{m} \vdash \mathtt{t}\ \mathtt{u} \hookrightarrow \mathtt{V},\ \mathtt{m'''}},$$

$$\frac{}{\mathtt{e},\ \mathtt{m} \vdash \mathbf{fun}\ \mathtt{x} \to \mathtt{t} \hookrightarrow \langle \mathtt{x}, \mathtt{t}, \mathtt{e} \rangle,\ \mathtt{m}},$$

$$\frac{}{\mathtt{e},\ \mathtt{m} \vdash \mathtt{n} \hookrightarrow \mathtt{n},\ \mathtt{m}},$$

$$\frac{\mathtt{e},\ \mathtt{m} \vdash \mathtt{u} \hookrightarrow \mathtt{q},\ \mathtt{m'} \qquad \mathtt{e},\ \mathtt{m}' \vdash \mathtt{t} \hookrightarrow \mathtt{p},\ \mathtt{m''}}{\mathtt{e},\ \mathtt{m} \vdash \mathtt{t} \otimes \mathtt{u} \hookrightarrow \mathtt{n},\ \mathtt{m''}}\ \text{если } \mathtt{p} \otimes \mathtt{q} = \mathtt{n},$$

$$\frac{\mathtt{e},\ \mathtt{m} \vdash \mathtt{t} \hookrightarrow \mathtt{0},\ \mathtt{m'} \qquad \mathtt{e},\ \mathtt{m}' \vdash \mathtt{u} \hookrightarrow \mathtt{V},\ \mathtt{m''}}{\mathtt{e},\ \mathtt{m} \vdash \mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v} \hookrightarrow \mathtt{V},\ \mathtt{m''}},$$

$$\frac{\mathtt{e},\ \mathtt{m} \vdash \mathtt{t} \hookrightarrow \mathtt{n},\ \mathtt{m'} \qquad \mathtt{e},\ \mathtt{m}' \vdash \mathtt{v} \hookrightarrow \mathtt{V},\ \mathtt{m''}}{\mathtt{e},\ \mathtt{m} \vdash \mathbf{ifz}\ \mathtt{t}\ \mathbf{then}\ \mathtt{u}\ \mathbf{else}\ \mathtt{v} \hookrightarrow \mathtt{V},\ \mathtt{m''}}\ \text{если } \mathtt{n} \text{ — ненулевое число},$$

$$\frac{(\mathtt{e}, \mathtt{x} = \langle \mathbf{fix}\ \mathtt{x}\ \mathtt{t}, \mathtt{e} \rangle),\ \mathtt{m} \vdash \mathtt{t} \hookrightarrow \mathtt{V},\ \mathtt{m'}}{\mathtt{e},\ \mathtt{m} \vdash \mathbf{fix}\ \mathtt{x}\ \mathtt{t} \hookrightarrow \mathtt{V},\ \mathtt{m'}},$$

$$\frac{\mathtt{e},\ \mathtt{m} \vdash \mathtt{t} \hookrightarrow \mathtt{W},\ \mathtt{m'} \qquad (\mathtt{e}, \mathtt{x} = \mathtt{W}),\ \mathtt{m}' \vdash \mathtt{u} \hookrightarrow \mathtt{V},\ \mathtt{m''}}{\mathtt{e},\ \mathtt{m} \vdash \mathbf{let}\ \mathtt{x} = \mathtt{t}\ \mathbf{in}\ \mathtt{u} \hookrightarrow \mathtt{V},\ \mathtt{m''}}.$$

Теперь можно привести правила для новых конструкций (**ref**, `!` и `:=`):

$$\frac{\mathtt{e},\ \mathtt{m} \vdash \mathtt{t} \hookrightarrow \mathtt{V},\ \mathtt{m'}}{\mathtt{e},\ \mathtt{m} \vdash \mathbf{ref}\ \mathtt{t} \hookrightarrow \mathtt{r},\ (\mathtt{m'}, \mathtt{r} = \mathtt{V})}\ \text{если } \mathtt{r} \text{ любая ссылка, не входящая в } \mathtt{m'},$$

$$\frac{\mathtt{e},\ \mathtt{m} \vdash \mathtt{t} \hookrightarrow \mathtt{r},\ \mathtt{m'}}{\mathtt{e},\ \mathtt{m} \vdash \mathtt{!t} \hookrightarrow \mathtt{V},\ \mathtt{m'}}\ \text{если } \mathtt{m'} \text{ содержит } \mathtt{r} = \mathtt{V},$$

$$\frac{\mathtt{e},\ \mathtt{m} \vdash \mathtt{t} \hookrightarrow \mathtt{r},\ \mathtt{m'} \qquad \mathtt{e},\ \mathtt{m}' \vdash \mathtt{u} \hookrightarrow \mathtt{V},\ \mathtt{m''}}{\mathtt{e},\ \mathtt{m} \vdash \mathtt{t} := \mathtt{u} \hookrightarrow \mathtt{0},\ (\mathtt{m''}, \mathtt{r} = \mathtt{V})}.$$

Конструкция `t; u`, семантика которой получается посредством интерпретации `t`, отбрасыванием вычисленного значения и последующей интерпретацией `u`, не слишком интересна в языке без побочных эффектов, потому что в этом случае значение терма `t; u` будет всегда таким же, что и значение `u`, при условии, что вычисление `t` завершается. Теперь же имеет смысл

добавить её в PCF:

$$\frac{\mathtt{e},\ \mathtt{m} \vdash \mathtt{t} \hookrightarrow \mathtt{V},\ \mathtt{m}' \qquad \mathtt{e},\ \mathtt{m}' \vdash \mathtt{u} \hookrightarrow \mathtt{W},\ \mathtt{m}''}{\mathtt{e},\ \mathtt{m} \vdash \mathtt{t};\ \mathtt{u} \hookrightarrow \mathtt{W},\ \mathtt{m}''}.$$

Сейчас можно также добавить конструкции **whilez**, **for** и др., которые не представляли никакого интереса в языке без побочных эффектов.

Упражнение 7.1. Напишите интерпретатор языка PCF со ссылками.

Упоминавшуюся в начале книги неопределённость, касающуюся вычисления вложенных функций, теперь можно окончательно разрешить.

Упражнение 7.2. Рассмотрим терм

let n = **ref** 0

in let f = **fun** x → **fun** y → x

in let g = **fun** z → (n := !n + z; !n) **in** f (g 2) (g 7)

(1) Каково значение этого терма? В каком порядке вычисляются аргументы в PCF? Почему?

(2) Измените приведённые выше правила таким образом, чтобы в результате вычисления этого терма получалось значение 2, а не 9.

(3) В разделе 2.5 мы отмечали: «В случае с применением...». Что вы думаете относительно этого замечания?

(4) Каково значение этого терма в Caml?

(5) Каково значение следующей программы на языке Java?

```
class Reference {
    static int n;
    static int f (int x, int y) {return x;}
    static int g (int z) {n = n + z; return n;}
    static public void main (String[ ] args) {
        n = 0;
        System.out.println(f(g(2),g(7)));
    }
}
```

(6) В каком порядке Caml и Java вычисляют свои аргументы?

Упражнение 7.3. Каково значение следующего терма, 10 или 11?

```
let x = ref 4
 in let f = fun y → y+!x
 in (x := 5; f 6)
```

Сравните ответ с упр. 2.8.

Упражнение 7.4. Напишите операционную семантику с большим шагом для конструкции **whilez**. Каково значение следующего терма?

```
let f = fun n →
   (let k = ref 1
    in let i = ref 1
    in (
      whilez (!i - n) do
        k := !k*!i;
        i := !i + 1
      done;
      !k))
 in f 3
```

Упражнение 7.5 (проблемы со ссылками при вызове по имени)**.** Рассмотрим приведённые выше правила операционной семантики ссылок с большим шагом.

(1) Какой стратегии они следуют, вызову по имени или вызову по значению? Дайте похожие правила для вызова по имени, сохранив **let** с вызовом по значению.

(2) Каково значение терма

```
let n = ref 0 in ((fun x → x + x) (n := !n + 1; 4)); !n
```

при вызове по значению и при вызове по имени?

(3) Каково значение терма

```
let n = ref 0 in ((fun x → 2 * x) (n := !n + 1; 4)); !n
```

при вызове по значению и при вызове по имени?

Упражнение 7.6 (типизация ссылок)**.** В целях типизации термов в расширении PCF ссылками добавим к языку типов символ **ref**, чтобы `nat ref` обозначало, к примеру, тип ссылки на натуральное число. Таким образом, если терм `t` имеет тип `A ref`, то `!t` это терм типа `A`.

(1) Добавьте к правилам типизации, приведённым в разделе 5.1, правила типизации ссылок.

(2) Напишите программу проверки типов для PCF со ссылками.

Комбинирование ссылок и полиморфизма — довольно тонкий вопрос, в этом упражнении мы не будем пытаться их смешать.

Упражнение 7.7 (от императивных программ к функциональным)**.** Рассмотрим терм `t`, определяющий функцию на натуральных числах с `p` аргументами и свободной переменной `n` типа `nat ref`. Сопоставим этому терму функцию с `p + 1` аргументом, которая возвращает пару натуральных чисел (см. упр. 3.13); образ $\mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{p}, \mathtt{m}$ это пара из значения терма **let** `n` = **ref** `m` **in** $(\mathtt{t}\ \mathtt{a}_1 \ldots \mathtt{a}_\mathtt{p})$ и значения терма `!n` в конце интерпретации.

(1) Какие функции будут сопоставлены следующим термам?

   а) **fun** `z` → `(n := !n + z; !n)`

   б) `(`**fun** `z` → `(n := !n + z; !n)) 7`

   в) `(`**fun** `x` → **fun** `y` → `x)`
   `((`**fun** `z` → `(n := !n + z; !n)) 2) ((`**fun** `z` → `(n := !n + z; !n)) 7)`

   Можно ли запрограммировать эти функции в PCF без ссылок?

   Попытайтесь проделать то же самое в общем виде.

(2) Какие функции будут сопоставлены следующим термам?

   а) **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_\mathtt{p} \to 2$

   б) **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_\mathtt{p} \to \mathtt{y}_1$

   в) **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_\mathtt{p} \to$`!n`

(3) Пусть `t` это терм типа `nat`, `f` — функция, сопоставленная терму **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_\mathtt{p} \to$ `t`. Какая функция будет сопоставлена терму **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_\mathtt{p} \to$ `n := t`?

(4) Пусть `t` и `u` это термы типа `nat`, `f` и `g` — функции, сопоставленные термам **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_\mathtt{p} \to$ `t` и **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_\mathtt{p} \to$ `u`. Какая функция будет сопоставлена терму **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_\mathtt{p} \to$ `(t + u)`?

(5) Пусть `t` и `u` это термы типа `nat`, `f` и `g` — функции, сопоставленные термам **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_p \to$ `t` и **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_p \to$ `u`. Какая функция будет сопоставлена терму **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_p \to$ `(t; u)`?

(6) Пусть `t` — терм типа $\mathtt{nat} \to \cdots \to \mathtt{nat} \to \mathtt{nat}$ с `q` аргументами типа `nat`, $\mathtt{u}_1, \ldots, \mathtt{u}_q$ — термы типа `nat`, $\mathtt{f}, \mathtt{g}_1, \ldots, \mathtt{g}_q$ — функции, сопоставленные термам **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_p \to$ `t`, **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_p \to \mathtt{u}_1$,..., **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_p \to \mathtt{u}_q$. Какая функция будет сопоставлена терму **fun** $\mathtt{y}_1 \to \cdots \to$ **fun** $\mathtt{y}_p \to (\mathtt{t}\, \mathtt{u}_1 \ldots \mathtt{u}_q)$?

(7) Можно ли запрограммировать эти функции в PCF без ссылок?

(8) Напишите программу, преобразующую терм PCF, содержащий эти символы и свободную переменную типа `nat ref`, в программу без ссылок, но с той же семантикой.

Упражнение 7.8 (для тех, кто предпочитает вместо `x := !x + 1` писать `x := x + 1`)**.** Рассмотрим конечное множество ссылок. Расширим PCF, введя для каждой из ссылок некоторую константу. Символ `:=` применяется теперь к изменяемой переменной и терму, что обозначается записью `X := t`.

Если `X` это изменяемая переменная, то значение, которое операционная семантика сопоставит терму `X`, будет значением, связанным со ссылкой `X`, в том состоянии, которое будет на момент интерпретации.

(1) Напишите для этого расширения PCF операционную семантику с большим шагом.

(2) Напишите интерпретатор для этого расширения PCF.

Упражнение 7.9 (минимальный императивный язык)**.** Рассмотрим язык с целочисленными константами, арифметическими операциями, изменяемыми переменными (см. упр. 7.8), присваиванием `:=`, последовательным выполнением `;`, условной операцией **ifz** и циклом **whilez** (но без традиционного понятия переменной, **fun**, **fix**, **let** и применения).

(1) Напишите правила, определяющие операционную семантику этого языка.

(2) Напишите для него интерпретатор.

(3) Запрограммируйте на этом языке вычисление факториала.

(4) Какие программы можно выразить средствами этого языка?

Завершая эту главу, заметим, что в большинстве языков программирования имеется два разных способа вычисления факториала. Например, в Java можно написать программу как рекурсивно:

```
static int fact(int x) {
    if (x == 0)
       return 1;
    return x * (fact (x - 1));
}
```

так и итеративно:

```
static int fact(int x) {
    int k = 1;
    for (int i = 1; i <= x; i = i + 1)
        k = k * i;
    return k;
}
```

Следует ли нам предпочесть первую версию второй или наоборот?

Разумеется, теория языков программирования не может дать ответа на вопросы типа «следует ли нам...». Мы, тем не менее, можем осветить те моменты, которые в связи с ним возникают.

В самых первых языках программирования — машинных языках, языках ассемблера, Фортране, Бейсике и др. — можно было запрограммировать только вторую версию. Действительно, программу с циклами и ссылками легче исполнять на машине, которая сама по сути является физической системой с изменяемым состоянием, нежели программу, требующую вычисления функции, определённой посредством неподвижной точки.

Одним из самых первых языков, поспособствовавшим использованию рекурсивных определений, был Lisp. Программы на нём впервые отошли от ссылок и побочных эффектов, что упростило семантику языка, приблизило его к математике, позволило программистам рассуждать о программах, доказывая их свойства, в более простых терминах, облегчило разработку сложных программ. Например, программу вычисления производной алгебраического выражения написать с помощью рекурсии гораздо проще, потому что не нужно хранить стек выражений, ждущих своей очереди на дифференцирование. В этих условиях было довольно естественно противопоставить чисто функциональный стиль программирования «нечистому» императивному.

Однако первые реализации функциональных языков по сравнению с императивными были довольно медленными, и прежде всего потому, что,

как уже было сказано, на машине как физической системе исполнять функциональную программу гораздо труднее, чем императивную. В 1990-е методы компиляции функциональных программ настолько развились, что сегодня аргументы, противопоставляющие функциональному программированию эффективность, более не действительны, возможно, за исключением высоконагруженных вычислительных приложений.

Более того, все современные языки программирования имеют как функциональные, так и императивные черты, поэтому единственным разумным аргументом, определяющим выбор конкретного стиля, сегодня остаётся простота и лёгкость использования.

С этой точки зрения становится очевидным, что далеко не все задачи одинаково хорошо подходят обоим стилям. Вычисление производных проще реализовать в функциональном стиле. Напротив, задавая программу для черепашки в Лого, гораздо более естественно говорить о положении черепашки и её ориентации — то есть о состоянии на данный момент. Столь же естественно рассуждать о действиях, выполняемых черепашкой: переместиться, нарисовать линию и пр. Довольно нелегко выразить всё это функционально, на самом деле совершенно неестественно представлять действия черепашки как функции на области рисования.

Остаётся ещё один таинственный момент: программы, будь то функциональные или императивные, это всегда функции из входных данных в выходные. Если императивное программирование предлагает новые способы определения функций, которые иногда более практичны с точки зрения информатики, чем определения в математике, типичные для функциональных языков, то можно задаться вопросом, а не могут ли они оказаться более практичными и для математиков? Однако понятие ссылки в математике всё ещё не нашло своего применения.

Глава 8. Записи и объекты

8.1. Записи

В уравнениях, которые описывают движение двух взаимодействующих тел, например, звезды и планеты, положение каждого задаётся тремя координатами (функциями от времени). Это приводит к системе дифференциальных уравнений с шестью неизвестными. Вместо независимого рассмотрения координат можно сгруппировать их по три, получив систему дифференциальных уравнений с векторными неизвестными. В математике имеются средства для группировки нескольких сущностей в одну: понятие пары, которое может быть применено многократно, чтобы получить кортеж, а также понятие конечной последовательности.

В программировании также требуются инструменты для подобной упаковки. Среди полезных в такой задаче средств можно выделить понятия *пары*, *массива*, *записи*, *объекта* и понятие *модуля*. Компоненты этих структур называются *полями*.

8.1.1. Помеченные поля

Чтобы обозначить положение предмета на Земле с помощью широты, долготы и высоты над уровнем моря, можно использовать кортеж с тремя компонентами, соответствующими этим параметрам. Если считать тройку `(a, b, c)` парой `(a, (b, c))`, то левая часть отвечает широте, левая компонента правой части соответствует долготе, а правая компонента правой части — высоте над уровнем моря. Можно было бы использовать другое сочетание, наш выбор здесь абсолютно произволен.

Если, с другой стороны, понимать тройку `(a, b, c)` как функцию из множества $\{0, 1, 2\}$ в $\mathbb{R}$, где `0` соответствует `a`, `1` относится к `b`, а `2` — к `c`, то широта, на которой находится предмет, это вещественное число, в которое данная функция переводит ноль, долготе отвечает значение в единице, а высоте над уровнем моря — значение в двойке. По-прежнему в нашем выборе остаётся известный произвол.

Нет надобности помещать эти поля кортежа в определённом порядке или ставить им в соответствие какие-либо числа. Более того, в таком случае, если бы в программе нам пришлось менять структуру данных и

добавлять новое поле в кортеж, мы столкнулись бы с необходимостью обновить текст программы в нескольких местах. Такие модификации подвержены ошибкам, и мы бы с лёгкостью могли спутать, к примеру, долготу и температуру.

Поскольку программистам удобней обращаться к полям по именам — «широта», «долгота» и т. п. — нежели по индексам или с использованием других чисел, языки программирования предоставляют такую возможность. Это приводит к понятию кортежа с помеченными полями, называемого «записью». С математической точки зрения запись это функция, область определения которой представляет собой некоторое конечное множество (вместо начального отрезка $\mathbb{N}$), элементами этого множества являются *метки* записи.

Идея обращения к полям по имени вместо позиции в кортеже может быть использована, кроме того, при вызове функций. В некоторых экспериментальных языках[1] вместо записи `f(4, 2)` следует писать `f(abscissa = 4, ordinate = 2)` или, что то же, `f(ordinate = 2, abscissa = 4)`.

8.1.2. Расширение PCF записями

Чтобы расширить PCF записями, добавим к языку три символа: создание записи обозначим $\{\}$, символ `.` позволит обращаться к отдельным полям записи, символ $\leftarrow$ используется для создания новой записи, идентичной исходной за исключением значения одного поля.

Перед тем, как точно определить эти символы, введём новый сорт для меток и бесконечное множество констант, по одной для каждой метки. Заметим, что символы для связывания переменной сорта метки отсутствуют, поэтому таких переменных в замкнутых термах не будет. Более того, язык не содержит никаких других символов, с помощью которых можно было бы получить терм сорта `label` («метка»), имеются только константы. Таким образом, в замкнутом терме единственные подтермы сорта `label` это константы. Теперь мы готовы добавить в PCF:

— символ $\{\}$ с `2n` аргументами, который не связывает переменные; на нечётных позициях в списке аргументов стоят метки, а на чётных термы;

— символ `.` с двумя аргументами, первый из которых это терм, а второй это метка; данный символ также не связывает переменные;

[1] В настоящее время «именованные аргументы функций» доступны, хотя необязательны к использованию, во многих промышленных языках, таких как C# 4.0, Common Lisp, Scala, PL/SQL, Python, R и Visual Basic. — *Прим. перев.*

— символ $\leftarrow$ с тремя аргументами, первый из которых это терм, второй это метка, а третий является термом; данный символ вновь не связывает переменные.

Упражнение 8.1. В определении языка из главы 1 каждый символ имел фиксированное число аргументов. Поэтому формально мы не имеем права ввести символ наподобие $\{\}$, который может иметь, к примеру, 6 или 8 аргументов. Как подправить определение, данное выше, чтобы снять противоречие с определением языка из главы 1? Подсказка: что такое список?

Терм $\{\}(\mathtt{l}_1, \mathtt{t}_1, \ldots, \mathtt{l}_n, \mathtt{t}_n)$ будет записываться как $\{\mathtt{l}_1 = \mathtt{t}_1, \ldots, \mathtt{l}_n = \mathtt{t}_n\}$, терм $.(\mathtt{t}, \mathtt{l})$ будем писать так: $\mathtt{t}.\mathtt{l}$, а терм $\leftarrow(\mathtt{t}, \mathtt{l}, \mathtt{u})$ представляется в виде $\mathtt{t}(\mathtt{l} \leftarrow \mathtt{u})$.

Операционная семантика с малым шагом для PCF будет включать теперь следующие правила:

$$\{\mathtt{l}_1 = \mathtt{t}_1, \ldots, \mathtt{l}_n = \mathtt{t}_n\}.\mathtt{l}_i \longrightarrow \mathtt{t}_i,$$

$$\{\mathtt{l}_1 = \mathtt{t}_1, \ldots, \mathtt{l}_n = \mathtt{t}_n\}(\mathtt{l}_i \leftarrow \mathtt{u}) \longrightarrow \{\mathtt{l}_1 = \mathtt{t}_1, \ldots, \mathtt{l}_{i-1} = \mathtt{t}_{i-1}, \mathtt{l}_i = \mathtt{u}, \mathtt{l}_{i+1} = \mathtt{t}_{i+1}, \ldots, \mathtt{l}_n = \mathtt{t}_n\}.$$

Аналогично операционная семантика с большим шагом расширяется правилами:

$$\frac{\mathtt{t}_1 \hookrightarrow \mathtt{V}_1 \quad \ldots \quad \mathtt{t}_n \hookrightarrow \mathtt{V}_n}{\{\mathtt{l}_1 = \mathtt{t}_1, \ldots, \mathtt{l}_n = \mathtt{t}_n\} \hookrightarrow \{\mathtt{l}_1 = \mathtt{V}_1, \ldots, \mathtt{l}_n = \mathtt{V}_n\}},$$

$$\frac{\mathtt{t} \hookrightarrow \{\mathtt{l}_1 = \mathtt{V}_1, \ldots, \mathtt{l}_n = \mathtt{V}_n\}}{\mathtt{t}.\mathtt{l}_i \hookrightarrow \mathtt{V}_i},$$

$$\frac{\mathtt{t} \hookrightarrow \{\mathtt{l}_1 = \mathtt{V}_1, \ldots, \mathtt{l}_n = \mathtt{V}_n\} \qquad \mathtt{u} \hookrightarrow \mathtt{W}}{\mathtt{t}(\mathtt{l}_i \leftarrow \mathtt{u}) \hookrightarrow \{\mathtt{l}_1 = \mathtt{V}_1, \ldots, \mathtt{l}_{i-1} = \mathtt{V}_{i-1}, \mathtt{l}_i = \mathtt{W}, \mathtt{l}_{i+1} = \mathtt{V}_{i+1}, \ldots, \mathtt{l}_n = \mathtt{V}_n\}}.$$

Отметим, что в этих правилах термы сорта `label` не интерпретируются. Так происходит, потому что, как упоминалось выше, они являются константами.

Упражнение 8.2. Создайте интерпретатор для PCF с записями.

Упражнение 8.3. Цель этого упражнения в том, чтобы представить черепашку Logo в виде записи, содержащей абсциссу, ординату и угол поворота. Черепашка должна обладать внутренним состоянием, чтобы оставаться

одним и тем же объектом несмотря на смену положения — см. введение к главе 7. Есть два варианта: черепашка может быть определена как запись ссылок на вещественные числа или как ссылка на запись вещественных чисел. Определите функцию `move-forward` (движение вперёд) в обоих случаях.

В этом задании предполагается наличие готового типа вещественных чисел и всех необходимых операций.

В таком определении семантики с большим шагом для PCF с записями интерпретация терма $\{\mathtt{a = 3 + 4, b = 2}\}$ требует выполнить сложение `3` и `4`. Как только значение $\{\mathtt{a = 7, b = 2}\}$ получено, доступ к полю `a` не требует арифметических действий.

Другой путь состоит в том, чтобы отложить вычисление суммы и рассматривать терм $\{\mathtt{a = 3 + 4, b = 2}\}$ как значение, которое может интерпретироваться в таком виде. В этом примере придётся интерпретировать терм `3 + 4` каждый раз при доступе к полю `a`. Можно сказать, что такая семантика является *вызовом по имени*, в отличие от семантики, данной выше, которая следует стратегии *вызова по значению*.

При вызове по имени правила операционной семантики выглядят следующим образом:

$$\frac{}{\{\mathtt{l_1 = t_1, \ldots, l_n = t_n}\} \hookrightarrow \{\mathtt{l_1 = t_1, \ldots, l_n = t_n}\}},$$

$$\frac{\mathtt{t} \hookrightarrow \{\mathtt{l_1 = t_1, \ldots, l_n = t_n}\} \qquad \mathtt{t_i} \hookrightarrow \mathtt{V}}{\mathtt{t.l_i} \hookrightarrow \mathtt{V}},$$

$$\frac{\mathtt{t} \hookrightarrow \{\mathtt{l_1 = t_1, \ldots, l_n = t_n}\}}{\mathtt{t(l_i \leftarrow u)} \hookrightarrow \{\mathtt{l_1 = t_1, \ldots, l_{i-1} = t_{i-1}, l_i = u, l_{i+1} = t_{i+1}, \ldots, l_n = t_n}\}}.$$

Упражнение 8.4. Создайте интерпретатор для PCF с записями, следуя семантике вызова по имени.

В сравнении двух семантик записей можно сделать те же замечания, что и при рассмотрении аналогичного вопроса для функций: интерпретация **let** `x` = $\{$`a = fact 10, b = 4`$\}$ **in** `x.b` требует вычисления факториала `10` при вызове по значению, но не при вызове по имени. С другой стороны, интерпретация **let** `x` = $\{$`a = fact 10, b = 4`$\}$ **in** `x.a + x.a` при вызове по имени потребует дважды вычислить факториал `10`. Интерпретация **let** `x` = $\{$`a =` **fix** `y y, b = 4`$\}$ **in** `x.b` в случае вызова по значению зацикливается, в то время как вызов по имени успешно завершается, вернув `4`.

Наконец, в присутствии ссылок побочный эффект от интерпретации поля может повторяться в случае многократного к нему обращения — см. упр. 7.5.

Рассмотрим, к примеру, запись `x` с полем `a`, которое представляет собой ссылку на натуральное число с начальным значением `0`; пусть, кроме того, функция `inc` увеличивает это число на единицу. Запишем терм, который увеличивает значение натурального числа и возвращает его:

```
let x = {a = ref 0}
 in let inc = fun y → (y.a := 1 + !(y.a))
 in (inc x; !(x.a))
```

При вызове по значению этот терм, как и ожидалось, сведётся к `1`. Однако интерпретация с вызовом по имени обратится к полю `a` записи `x` трижды, то есть три раза в интерпретации встретится терм **ref** `0`, что создаст три ссылки на значение `0`. Третья ссылка, созданная при интерпретации `!(x.a)` никогда не изменяется, таким образом, при вызове по имени терм даст в результате `0`.

Чтобы обеспечить одинаковые результаты при вызове по имени и вызове по значению, нужно исключить побочные эффекты — такие как создание ссылки в примере выше — при интерпретации полей. Можно переписать предшествующий терм следующим образом:

```
let r = ref 0
 in let x = {a = r}
 in let inc = fun y → (y.a := 1 + !(y.a))
 in (inc x; !(x.a))
```

В этом случае результат будет равен `1`, какой бы порядок вызова мы ни использовали.

Упражнение 8.5 (типы для записей)**.** Рассмотрим тип `person` для записей с тремя полями: `surname`, `name` и `telephone`. Покажите, что можно запрограммировать три функции: `x(surname ← y)`, `x(name ← y)` и `x(telephone ← y)` без использования символа ←, что окажется свидетельством его избыточности.

Будет ли этот символ по-прежнему избыточным в присутствии типа `contactable` для записей, содержащих поле `telephone`?

Сохраняется ли единственность типов при наличии типов `person` и `contactable`?

8.2. Объекты

Обычно программы имеют дело с разнообразными типами, которые часто представляются записями. Например, компьютерная система компании может обрабатывать формы заказов от покупателей, счета-фактуры, денежные переводы и т. п. Заказ покупателя может быть представлен в виде записи, в которую входят идентификатор выбранного товара, количество и т. д. Есть несколько подходов к предоставлению возможности печати этих данных. Можно создать единственную функцию `print`, которая в начале своей работы определяет, какой вид данных требуется распечатать. С другой стороны, мы могли бы написать несколько различных функций: `print_order_form`, `print_pay_slip` и т. п. Ещё можно было бы определить запись `print`, каждое поле которой станет функцией печати. Другой вариант заключается в том, чтобы сделать функцию печати частью нужного типа. Полученный таким образом тип данных называется классом, а элементы этого типа — *объектами*.

В наиболее радикальных подходах к объектно-ориентированному программированию каждый объект, например, каждая форма заказа, включает собственную версию функции `print`. Таким образом, форма заказа представляет из себя запись, которая в дополнение к обычным полям — идентификатору нужного товара, количеству единиц товара и т. д. — содержит поле `print`, определяющее функцию, которая должна использоваться для печати данного объекта.

В некоторых языках программирования, например, в Java, функция `print` ставится в соответствие каждому классу, а не каждому объекту. В этом случае все объекты класса используют одну функцию печати, определённую статически или динамически. Если два объекта `t` и `u` класса `C` должны использовать разные функции печати, необходимо определить два подкласса `T` и `U` класса `C`; подклассы будут наследовать все поля класса `C`, но переопределят функцию `print` по-своему.

8.2.1. Методы и функциональные поля

Объект это обычная запись, в которой некоторые поля являются функциями. В Java, где функции не принадлежат к числу объектов первого класса, следует разделять поля, которые являются функциями, и все прочие; функциональные поля называются *методами*.

В языке, где функции представляют собой объекты первого класса, как в PCF, такое разделение не обязательно. Объекты сводятся к записям, и объектно-ориентированный стиль доступен в расширении PCF с записями, которое было определено в предыдущем разделе.

Упражнение 8.6. Программа, которая управляет продажей билетов на концерт, представляется объектом со следующими полями:

— ссылка на натуральное число: количество билетов в партере;

— ссылка на натуральное число: количество билетов на балконе;

— функция, которая принимает объект и натуральное число в качестве аргументов — `0` для партера и `1` для балкона — и возвращает `1` или `0` в зависимости от того, остались или нет свободные места в данной части зала соответственно;

— функция, которая принимает объект и натуральное число в качестве аргументов — `0` для партера и `1` для балкона — и бронирует место, уменьшая количество мест, оставшихся в данной части зала; в соответствии с соглашением функция возвращает `0`.

Запрограммируйте данный объект на PCF с записями.

Системы типов для записей и объектов выходят за рамки рассмотрения данной книги. Скажем лишь, что, если приписать тип `A` объекту из упр. 8.6, то `A` должен быть декартовым произведением **ref** `nat`, **ref** `nat`, `A → nat → nat` и `A → nat → nat`. Однако мы не можем определить тип `A` как `(`**ref** `nat)` × `(`**ref** `nat)` × `(A → nat → nat)` × `(A → nat → nat)`, поскольку такое определение оказалось бы циклическим. Чтобы задать такой тип, нужно ввести оператор неподвижной точки на типах.

Если `X → Y` определяет пространство функций из `X` в `Y`, а `B` это множество с не менее чем двумя элементами, то рекурсивное уравнение `A = (A → B)` не имеет решений. Действительно, теорема Кантора говорит о том, что мощность множества `A → B` строго больше, чем `A`. Уравнение `A = (`**ref** `nat)` × `(`**ref** `nat)` × `(A → nat → nat)` × `(A → nat → nat)` также не имеет решений. Как и в случае с конструкцией **fix**, задать денотационную семантику для оператора неподвижной точки на типах непросто.

8.2.2. Что значит «Self»?

Пусть `t` это объект, построенный в упр. 8.6. Чтобы узнать, остались ли свободные места в партере, необходимо интерпретировать терм `t.free t 0`. Действительно, функция `t.free` принимает объект `u` и натуральное число `n` и говорит о том, равно ли нулю поле в `u`, связанное с `n` — количеством мест в партере для `n = 0` или на балконе для `n = 1`. Другими словами, метод `free` является *статическим* в терминологии Java.

Хотелось бы иметь возможность непосредственно применять метод `free` объекта `t` к самому объекту `t`, то есть писать `t#free 0` вместо

`t.free t 0`. Другими словами, этот метод должен быть *динамическим*.

Один из подходов к этому состоит в том, чтобы рассматривать терм `t#l` как сокращённую запись `t.l t`. Сложность здесь кроется в следующем: если `t` это объект и `l` представляет метку в нём, то терм `t#l` можно использовать только в том случае, если `l` это функция типа $\mathtt{A} \to \ldots$, где `A` это тип самого `t`. Другими словами, терм `t#l` можно использовать только тогда, когда `l` это метка метода. Если `l` это метка поля, которое не является методом, по-прежнему нужно писать `t.l`.

Чтобы избежать такого разделения, можно сделать все поля функциями. Если поле `a` объекта `t` имеет значение `3`, превратим его в функциональное значение $\mathbf{fun}\ \mathtt{s} \to \mathtt{3}$. Тогда терм `t#a`, то есть `t.a t` или $(\mathbf{fun}\ \mathtt{s} \to \mathtt{3})\ \mathtt{t}$ интерпретируется как значение `3`.

Первый аргумент каждого метода в этом случае является связанной переменной, которая обычно называется `this` или `self`. Фактически большинство языков программирования используют специальную переменную `self` или `this`, которая неявно связывается внутри объекта и обозначает сам этот объект.

Когда все методы в записи это термы вида $\mathbf{fun}\ \mathtt{x} \to \ldots$, они могут быть интерпретированы независимо, и в этом случае можно упростить правило

$$\frac{\mathbf{fun}\ \mathtt{x}_1 \to \mathtt{t}_1 \hookrightarrow \mathtt{V}_1 \ldots}{\{\mathtt{l}_1 = \mathbf{fun}\ \mathtt{x}_1 \to \mathtt{t}_1, \ldots\} \hookrightarrow \{\mathtt{l}_1 = \mathtt{V}_1, \ldots\}},$$

используя вместо него следующее:

$$\frac{}{\{\mathtt{l}_1 = \mathbf{fun}\ \mathtt{x}_1 \to \mathtt{t}_1, \ldots\} \hookrightarrow \{\mathtt{l}_1 = \mathbf{fun}\ \mathtt{x}_1 \to \mathtt{t}_1, \ldots\}}.$$

Аналогично правило

$$\frac{\mathtt{t} \hookrightarrow \{\mathtt{l}_1 = \mathtt{V}_1, \ldots\}}{\mathtt{t.l_i} \hookrightarrow \mathtt{V_i}}$$

заменяется правилом

$$\frac{\mathtt{t} \hookrightarrow \{\mathtt{l}_1 = \mathbf{fun}\ \mathtt{x}_1 \to \mathtt{t}_1, \ldots\}}{\mathtt{t.l_i} \hookrightarrow \mathbf{fun}\ \mathtt{x_i} \to \mathtt{t_i}}.$$

Наконец, правило

$$\frac{\mathtt{t} \hookrightarrow \{\mathtt{l}_1 = \mathtt{V_i}, \ldots\} \qquad \mathtt{u} \hookrightarrow \mathtt{W}}{\mathtt{t(l_i \leftarrow u)} \hookrightarrow \{\mathtt{l}_1 = \mathtt{V}_1, \ldots, \mathtt{l_i} = \mathtt{W}, \ldots\}}$$

может быть заменено на

$$\frac{\mathtt{t} \hookrightarrow \{\mathtt{l_1} = \textbf{fun}\ \mathtt{x_1} \to \mathtt{t_1}, \ldots\}}{\mathtt{t(l_i} \leftarrow \mathtt{(fun\ y} \to \mathtt{v))} \hookrightarrow \{\mathtt{l_1} = \textbf{fun}\ \mathtt{x_1} \to \mathtt{t_1}, \ldots, \mathtt{l_i} = \mathtt{(fun\ y} \to \mathtt{v)}, \ldots\}}.$$

Чтобы все поля действительно были функциями, можно модифицировать язык записей, перейдя к объектно-ориентированному языку. Символ {} теперь будет связывать одну переменную в каждом чётном аргументе (терме), символ . заменяется на #, а символ ← связывает переменную в третьем аргументе.

Терм $\{\}(\mathtt{l_1}, \mathtt{s_1}\ \mathtt{t_1}, \ldots, \mathtt{l_n}, \mathtt{s_n}\ \mathtt{t_n})$ записывается следующим образом: $\{\mathtt{l_1} = \zeta \mathtt{s_1}\ \mathtt{t_1}, \ldots, \mathtt{l_n} = \zeta \mathtt{s_n}\ \mathtt{t_n}\}$, терм $\mathtt{\#(t, l)}$ записывается как $\mathtt{t\#l}$, а терм $\leftarrow \mathtt{(t, l, s\ u)}$ будет выглядеть как $\mathtt{t(l} \leftarrow \zeta \mathtt{s\ u)}$. Запишем правила операционной семантики с большим шагом в новом виде:

$$\frac{}{\{\mathtt{l_1} = \zeta \mathtt{s_1}\ \mathtt{t_1}, \ldots, \mathtt{l_n} = \zeta \mathtt{s_n}\ \mathtt{t_n}\} \hookrightarrow \{\mathtt{l_1} = \zeta \mathtt{s_1}\ \mathtt{t_1}, \ldots, \mathtt{l_n} = \zeta \mathtt{s_n}\ \mathtt{t_n}\}},$$

$$\frac{\mathtt{t} \hookrightarrow \{\mathtt{l_1} = \zeta \mathtt{s_1}\ \mathtt{t_1}, \ldots, \mathtt{l_n} = \zeta \mathtt{s_n}\ \mathtt{t_n}\} \qquad \mathtt{(t/s_i)t_i} \hookrightarrow \mathtt{V}}{\mathtt{t\#l_i} \hookrightarrow \mathtt{V}},$$

$$\frac{\mathtt{t} \hookrightarrow \{\mathtt{l_1} = \zeta \mathtt{s_1}\ \mathtt{t_1}, \ldots\}}{\mathtt{t(l_i} \leftarrow \zeta \mathtt{s\ u)} \hookrightarrow \{\mathtt{l_1} = \zeta \mathtt{s_1}\ \mathtt{t_1}, \ldots, \mathtt{l_{i-1}} = \zeta \mathtt{s_{i-1}}\ \mathtt{t_{i-1}}, \mathtt{l_i} = \zeta \mathtt{s\ u}, \mathtt{l_{i+1}} = \zeta \mathtt{s_{i+1}}\ \mathtt{t_{i+1}}, \ldots\}}.$$

Упражнение 8.7. Создайте интерпретатор языка PCF с объектами.

Упражнение 8.8 (позднее связывание)**.** Рассмотрим терм

$$\mathtt{(}\mathtt{(\{x} = \zeta \mathtt{s\ 4, f} = \zeta \mathtt{s\ fun\ y} \to \mathtt{y + s\#x\}(x} \leftarrow \zeta \mathtt{s\ 5))\#f)6}$$

Каково значение этого терма, 10 или 11? Сравните это с результатом упр. 2.8.

8.2.3. Объекты и ссылки

Стандартное определение объекта включает понятие внутреннего состояния, которое меняется со временем. Таким образом, оно сочетает понятия объекта и ссылки, которые, очевидно, разделены в функциональных объектах, введённых выше.

В языке с объектами и ссылками, где нефункциональное поле `a = u` преобразовано в `a = fun x → u`, интерпретация **fun** `x` → `u` не имеет побочных эффектов, которые получаются при интерпретации `u`. Побочные эффекты возникают только при доступе к полю. Получается, что поведение в данном случае схоже с семантикой вызова по имени для записей. Терм

```
let x = {a = fun s → ref 0}
 in let inc = fun s → (s#a := 1 + !(s#a))
 in (inc x; !(x#a))
```

интерпретируется как `0`, а не как `1`, аналогично терму

```
let x = {a = ref 0}
 in let inc = fun s → (s.a := 1 + !(s.a))
 in (inc x; !(x.a))
```

при вызове по значению. Следует переписать этот терм по-другому:

```
let r = ref 0
 in let x = {a = fun s → r}
 in let inc = fun s → (s#a := 1 + !(s#a))
 in (inc x; !(x#a))
```

если желательно иметь `1` в результате его интерпретации.

Упражнение 8.9. Сколько раз во время интерпретации терма `t#l` интерпретируется терм `t`? Если интерпретация `t` имеет побочные эффекты, сколько раз они произойдут? Как можно обеспечить однократную интерпретацию терма `t`?

Послесловие

Первая среди целей этой книги состояла в том, чтобы представить основные средства определения семантики языка программирования: операционную семантику с малым шагом, операционную семантику с большим шагом и денотационную семантику.

Мы обратили особое внимание на то, что все три инструмента служат одной цели: определить отношение $\hookrightarrow$, связывающее программу, входные данные и результат. Поскольку задача предполагает определение отношения, естественно задать вопрос: как определяются отношения на математическом языке.

Ответ одинаков во всех трёх случаях: цель достигается с помощью теорем о неподвижной точке. Однако такая схожесть поверхностна: теоремы о неподвижной точке используются в трёх перечисленных подходах к определению семантики по-разному. Эти теоремы играют важную роль в операционной семантике, так как открывают доступ к индуктивным определениям и рефлексивно-транзитивным замыканиям. Напротив, в денотационной семантике теоремам о неподвижной точке отводится второстепенная роль, потому что там они используются лишь для определения смысла конструкции **fix**. Денотационная семантика языка без оператора неподвижной точки, например, System T (см. упр. 5.13), может быть определена без использования теорем о неподвижной точке.

Лучше увидеть различия рассмотренных подходов к заданию семантики можно, обратив внимание на роль вывода. Чтобы утверждать, что терм `t` имеет значение `V` в операционной семантике, достаточно предъявить соответствующий вывод, или последовательность редукций, то есть конечный объект. Напротив, в денотационной семантике значение терма вида **fix** задаётся как неподвижная точка некоторой функции, то есть как предел. По этой причине, для того, чтобы установить, что значением терма `t` является `V`, иногда приходится вычислять предел последовательности, то есть время от времени мы сталкиваемся с бесконечными объектами.

Операционная семантика имеет преимущество перед денотационной в том, что отношение $\hookrightarrow$ может быть определено более конструктивно. С другой стороны, операционно можно определить лишь рекурсивно перечислимые отношения, в то время как денотационно определяются произвольные отношения. По этой причине в операционной семантике нельзя пополнить

определение отношения $\hookrightarrow$, приписав значение $\bot$ термам, которые не завершаются, потому что получившееся отношение не является рекурсивным, оно не может быть эффективно определено по индукции. Напротив, ничто не мешает добавить такой символ в контексте денотационной семантики.

Мы видим дилемму, вызванную неразрешимостью проблемы останова: нельзя пополнить отношение $\hookrightarrow$, добавив в него $\bot$ для незавершающихся термов, и одновременно определять его индуктивно. Приходится выбирать между пополнением и индуктивным определением отношения, что приводит к двум различным семантикам. Читатель, посещавший курсы по логике, заметит здесь те же вопросы, что разделяли истинные высказывания на те, которые определялись индуктивно через существование доказательства, и те, которые проверялись на выполнимость в некоторой модели.

Второй целью книги было определение семантики для нескольких стандартных конструкций языков программирования: явно определённых функций, функций, заданных как неподвижная точка, присваивания, записей, объектов и т. д. Вновь, если речь идёт об определении функций, полезным оказывается рассмотрение средств, с помощью которых определяются функции в математике. В целом, сравнение математического языка и языков программирования плодотворно, поскольку математический язык это самое близкое к языкам программирования из того, что мы знаем. Такое сравнение выявляет как общие моменты, так и некоторые различия.

Рассмотрение стандартных конструкций языков программирования здесь не является исчерпывающим, но даёт несколько содержательных примеров. Стоит помнить, что как зоология не изучает все виды животных один за одним, теория языков программирования не должна сводиться к изучению всех подряд языков программирования. Последние должны быть систематизированы в соответствии со своими основными чертами.

Мы могли бы продолжить наше исследование определением типов данных и исключений. Первое дало бы возможность снова применить теорему о неподвижной точке и алгоритм унификации Робинсона, частным случаем которого является задача сопоставления. Продолжая изучение в этом направлении, можно было бы познакомиться с понятием отката (backtracking), которое привело бы нас к языку программирования Пролог. Другие важные моменты, оставшиеся не рассмотренными, это полиморфная типизация ссылок, понятие массива, императивные объекты, модули, системы типов для записей и объектов (в том числе понятие подтипа), параллелизм и др.

Последняя цель этой книги была в демонстрации приложений перечисленных средств, в частности, для проектирования и реализации интерпретаторов и компиляторов, а также систем вывода типов. Основная мысль

здесь в том, что структура компилятора выводится напрямую из операционной семантики языка, который требуется компилировать. Следующим шагом стало бы изучение методов реализации абстрактных машин, а это, в свою очередь, привело бы к рассмотрению задач управления памятью и сборки мусора. Можно было бы также изучить анализ программ и построение систем вывода свойств программ в автоматическом или интерактивном режиме, например, для свойства, которое говорит о том, что значение, возвращаемое алгоритмом сортировки, является отсортированным списком.

Последняя тема, которую осталось обсудить, это роль теории языков программирования, в частности, вопрос: сводится ли её задача к описанию существующих языков программирования или к предложению новых языков.

Астрономы изучают существующие галактики и не строят новых, в то время как химики исследуют существующие молекулы и создают новые. Известно, что в последнем случае порядок появления теорий и промышленных методов может быть различным: преобразование массы в энергию стали осуществлять намного позже формулировки теории относительности, в то время как паровой двигатель появился задолго до открытия принципов термодинамики.

Теория языков программирования открыла путь к разработке новых свойств, таких как статическое связывание, вывод типов, полиморфные типы, сборка мусора и т. д., которые теперь доступны в промышленных языках программирования. Напротив, другие средства, такие как присваивание и объекты, были введены в языки программирования спонтанно, и теория здесь опоздала. Разработка формальной семантики для этих конструкций привела к новым идеям, таким как недавнее расширение Java полиморфными типами[2].

Теория языков программирования не занимается исключительно описанием или постановкой новых целей. Именно постоянное переключение между изучением существующих средств и разработкой новых придаёт теории языков программирования динамику.

[2]Имеется в виду включение в язык Java (J2SE 5.0, 2004 год) обобщённых типов (*generics*). — *Прим. перев.*

Библиография

1. Abadi M., Cardelli L. A Theory of Objects. Springer, Berlin (1998).
2. Dybvig R.K. The Scheme Programming Language, 2nd edn. Prentice Hall, New York (1996). www.scheme.com/tspl2d/
3. Gunter C.A. Semantics of Programming Languages: Structures and Techniques. MIT Press, Cambridge (1992).
4. Kahn G. Natural semantics. In: Proceedings of the Symp. on Theoretical Aspects of Computer Science, TACS, Passau (1987).
5. Mitchell J.C. Foundations for Programming Languages. MIT Press, Cambridge (1996). [*Русский перевод*: Митчелл Дж. Основания языков программирования. М.–Ижевск: НИЦ «Регулярная и хаотическая динамика», 2010.]
6. Mitchell J.C. Concepts in Programming Languages. Cambridge University Press, Cambridge (2002).
7. Peyton Jones S., Lester D. Implementing functional languages: a tutorial. Prentice Hall, New York (1992).
8. Pierce B.C. Types and Programming Languages. MIT Press, Cambridge (2002). [*Русский перевод*: Пирс Б. Типы в языках программирования. М.: Лямбда пресс, Добросвет, 2012.]
9. Plotkin G.D. LCF considered as a programming language. Theor. Comput. Sci. 5, 223–255 (1977).
10. Plotkin G.D. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.
11. Reynolds J.C. Theories of Programming Languages. Cambridge University Press, Cambridge (1998).
12. Scott D. Continuous Lattices. Lecture Notes in Math., vol. 274, pp. 97–136. Springer, Berlin (1972).
13. Weis P., Leroy X. Le langage Caml, 2nd edn. Dunod, Paris (1999).
14. Winskel G. The Formal Semantics of Programming Languages. MIT Press, Cambridge (1993).

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» по электронному адресу orders@alians-kniga.ru.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон.

Эти книги вы можете заказать и в Internet-магазине: www.dmk-press.ru.

Оптовые закупки: электронный адрес books@alians-kniga.ru.

Жиль Довек, Жан-Жак Леви

Введение в теорию языков программирования

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Перевод с английского *Брагилевский В. Н.,*
Пеленицын А. М.
Вёрстка *Брагилевский В. Н.,*
Пеленицын А. М.
Корректор *Синяева Г. И.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 05.02.2013. Формат 60×90 1/16.
Вёрстка выполнена средствами TEXLive 2012.
Гарнитура «Computer Modern». Печать офсетная.
Усл. печ. л. 8,5. Тираж 200 экз.

Издательство ДМК Пресс
Электронный адрес издательства: www.dmk-press.ru